# Implémentation et évaluation d'algorithmes parallèles de FFTs 3D à base de modèles de composants logiciels

Jérôme Richard

▶ **To cite this version:**

Jérôme Richard. Implémentation et évaluation d'algorithmes parallèles de FFTs 3D à base de modèles de composants logiciels. Calcul parallèle, distribué et partagé [cs.DC]. 2014. hal-01082575

**HAL Id: hal-01082575**
**https://inria.hal.science/hal-01082575**

Submitted on 17 Nov 2014

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

Master 2 - Visualisation Image Performance
Université d'Orléans (2013-2014)

# Implémentation et évaluation d'algorithmes parallèles de FFTs 3D à base de modèles de composants logiciels



Jérôme RICHARD

Encadrants : Vincent LANORE et Christian PEREZ
Rapporteur universitaire : Sophie ROBERT

# Table des matières

# Remerciements

Je tiens tout d'abord à remercier mes tuteurs de stages Christian Perez et Vincent Lanore, membres de l'équipe Avalon du Laboratoire d'Informatique du Parallélisme (LIP) qui m'ont offert la possibilité de travailler sur la thématique de mon stage et m'ont suivi tout au long de celui-ci.

Je remercie également Laurent Pouilloux pour m'avoir aidé à développer des outils d'automatisation et à configurer la plate-forme expérimentale Grid'5000.

Je tiens aussi à remercier les autres membres d'Avalon qui m'ont accueilli au sein de l'équipe, avec qui je me suis bien entendu et qui m'ont permis de travailler dans un environnement agréable durant toute la durée de mon stage.

Enfin, je remercie l'ensemble des membres du laboratoire ainsi que ceux qui m'ont permis d'effectuer mes expériences dans de bonnes conditions sur les supercalculateurs Curie et Jade ainsi que sur Grid'5000.

# 1 Environnement et outils du stage

Afin de valider son Master Recherche, chaque étudiant du Département Informatique de l'UFR Science d'Orléans doit accomplir un stage de six mois. Cette expérience doit permettre de mettre en application les connaissances acquises durant la formation et de s'initier au monde de la recherche. J'ai pour ma part effectué un stage au Laboratoire d'Informatique du Parallélisme (LIP) de Lyon.

Le LIP est un laboratoire de recherche en informatique en association avec le CNRS, l'ENS de Lyon et l'Université Claude Bernard Lyon 1 ainsi que Inria. Le laboratoire est structuré en sept équipes :

- AriC (arithmetique et calculs)
- Avalon (algorithmes et architectures logicielles pour les plates-formes distribuées et HPC)
- Compsys (compilation, systèmes embarqués)
- DANTE (réseaux dynamiques)
- MC2 (modèles de calcul, complexité et combinatoire)
- PLUME (programmes et preuves)
- ROMA (modèles, algorithmes et ordonnancement)

L'équipe Avalon, sous la direction de Christian Perez, s'intéresse plus spécifiquement aux algorithmes et aux architectures logicielles pour des plates-formes distribuées et à haute performance. Elle traite de nombreux sujets tels que l'algorithmique distribuée, les modèles de programmation, le composition et la déploiement de services, le traitement de gros volumes de données... C'est dans cette équipe que j'ai été accueilli pour effectuer mon stage.

J'ai eu l'opportunité d'avoir un accès à Grid'5000 durant mon stage. Grid'5000 [1] est une plate-forme expérimentale constituée d'un ensemble de sites (tels que Grenoble, Lyon, Rennes, etc.) reliés entre eux. Chaque site dispose d'un ensemble de clusters eux-mêmes interconnectés via un réseau rapide et contenant des ressources de calcul la plupart du temps homogènes. Cette plate-forme m'a permis d'effectuer de nombreux tests applicatifs (développement, passage à l'échelle, tests de variation des architectures).

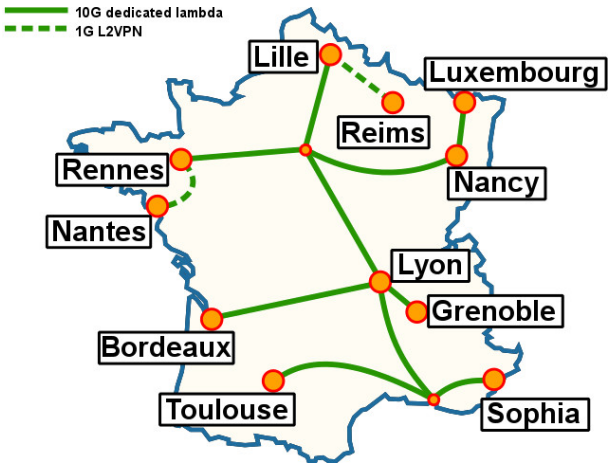



FIGURE 1.1 – Topologie des sites de Grid'5000

J'ai aussi eu la chance de disposer d'heures de calcul sur le supercalculateur Curie (nœuds fins) du TGCC via l'appel DARI (Demande d'Attribution de Ressources Informatiques) dans le cadre du projet européen PRACE-2IP, visant à évaluer de nouvelles architectures, des technologies, des

systèmes et des logiciels de calcul haute performance (HPC). Cette allocation d'heures m'a permis de tester le passage à l'échelle de mes applications sur une architecture distribuée disposant d'un grand nombre de cœurs.

# 2 Introduction

Ces dernières années, la complexification des modèles scientifiques et l'augmentation de la taille et de la précision des domaines de calcul ont considérablement accru les besoins en puissance de calcul (médecine, météorologie, fusion nucléaire, etc.).Parallèlement, la puissance de calcul des machines suit une croissance exponentielle. L'accroissement des performances des ordinateurs a longuement été possible via augmentation de la fréquence des processeurs. Or, l'augmentation de la fréquence n'est aujourd'hui plus possible, car elle engendrerait une augmentation trop grande de la chaleur dissipée et de la consommation des unités de calculs. Pour pallier ces limitations, les processeurs se sont vus dotés de plusieurs cœurs et les ordinateurs parfois de plusieurs processeurs. Dans le cas des supercalculateurs, le nombre de machines interconnectées a été démultiplié afin de faire face à la demande croissante en puissance de calcul.

Les architectures distribuées (supercalculateurs, grappe de serveurs, etc.) sont complexes et diversifiées. Par exemple, la façon d'interconnecter les nœuds de calcul peut différer d'un supercalculateur à l'autre. Certains d'entre eux, tel que le K computer, disposent d'un réseau d'interconnexion spécifique (qui n'est présent que sur ce celui-ci). La présence d'accélérateurs de calcul ou de techniques d'optimisations matérielles misent en place par certains supercalculateurs (tels que ceux de la famille Cray XTs ou Blue Gene/Q) confirme la grande variabilité des architectures matérielles des supercalculateurs. Cela est aussi vrai dans le cas des grilles de calcul qui est par nature l'hétérogène, ou même pour des clusters.

Afin que les applications exploitent pleinement ces architectures matérielles, elles doivent implémenter des algorithmes adaptés à l'architecture matérielle sous-jacente. En effet, la grande diversité de ces architectures rend difficile l'utilisation d'un unique algorithme qui serait adapté à toutes les architectures matérielles. Dans le cadre de ce stage, nous nous sommes intéressées spécifiquement au calcul de transformée de Fourier rapide (FFT) sur ces architectures.

La FFT est un algorithme efficace pour calculer la transformée de Fourier discrète (DFT), une opération mathématique de traitement du signal utilisée dans de nombreux domaines scientifiques (*e.g.* dynamique moléculaire, traitement du signal, météorologie, etc.). La FFT est utilisée comme base dans de nombreuses applications scientifiques à haute performance traitant de gros volumes de données et ciblant des architectures distribuées. Le temps de calcul de FFTs distribuées constitue la plupart du temps une part non négligeable du temps de calcul total de ces applications.

Ainsi, de nombreuses recherches ont été réalisées concernant l'implémentation de FFTs distribuées et des optimisations ont été proposées. Ces optimisations tirent avantages d'architectures matérielles spécifiques ou de la valeur de certains paramètres d'entrée du calcul de la FFT (tels que la taille des données). Mais vu la rapidité d'évolution des architectures matérielles, de nouvelles optimisations sont sans cesse conçues. Par conséquent, les codes de FFTs doivent souvent être adaptés pour prendre en considération les évolutions et ainsi maximiser les performances.

L'adaptation de codes de FFTs à un usage spécifique a un coût en terme de temps de développement et nécessite une bonne connaissance de l'architecture cible et des FFTs. Cela peut être une tâche ardue pour une personne autre que celle qui a écrit le code original. À moins d'être automatisée, l'adaptation des codes de FFTs pour des usages spécifiques est souvent trop coûteuse. Des bibliothèques existantes (*e.g.* FFTW [2], OpenMPI MCA [3]) mettent en place une forme d'adaptation automatique, mais nous n'en n'avons trouvé aucune qui permette d'intégrer facilement de nouvelles optimisations.

Une solution prometteuse, actuellement étudiée, pour manipuler facilement la structure des algorithmes de FFTs et leur spécialisation consiste à utiliser des techniques de génie logiciel à base de composants [4]. Cette approche propose de construire une application en assemblant des unités

logicielles, nommées composants, dotées d'interfaces bien définies. La syntaxe et la sémantique des interfaces et de l'assemblage sont fixées par un modèle de composants. Une telle approche permet de facilement réutiliser les composants (pouvant provenir d'autres applications) et de disposer d'un haut niveau d'adaptation via l'assemblage. De plus, certains modèles de composants [5, 6] et outils permettent de générer ou optimiser automatiquement les assemblages.

Parmi l'ensemble des modèles de composants, seulement un sous ensemble très restreint propose un surcoût à l'exécution suffisamment faible pour pouvoir implémenter des applications scientifiques à haute performance en minimisant l'impact du modèle à exécution. Parmi eux, on peut trouver $L^2C$ [7], un modèle de composants de bas niveau à haute performance basé sur C++/FORTRAN et MPI [8] (Message Passing Interface).

La contribution de ce stage est d'implémenter des algorithmes de FFTs en trois dimensions existants avec le modèle de composants de haute performance $L^2C$ en fonction de paramètres fixés.

Dans le chapitre 3, nous présentons les architectures parallèles, leur classification, leur constitution ainsi que la manière dont on programme dessus. Puis dans le chapitre 4, nous expliquons ce que sont les FFTs 3D et énonçons les travaux qui y sont liés, en particulier quelques algorithmes connus, optimisations et bibliothèques existantes. Ensuite, au sein du chapitre 5, nous traitons des modèles de composants et donnons un aperçu de quelques modèles distribués ainsi que des détails sur $L^2C$ et HLCM. Dans le chapitre 6, nous décrivons de multiples assemblages conçus et implémentés avec $L^2C$ permettant de calculer des FFTs 3D de différentes manières. Le chapitre 7, nous comparons les implémentations des assemblages présentés avec des bibliothèques de FFTs en terme de performances, de réutilisation et de facilité d'adaptation. Enfin, le chapitre 8 conclut ce rapport et fournit quelques perspectives.

# 3 Architectures parallèles

## 3.1 Classification

Il existe plusieurs manières de classifier les architectures parallèles. Introduite 1966, la taxonomie de Flynn [9] vise à analyser le nombre de flots d'instructions et de flots de données des architectures. Cette taxonomie propose de diviser les architectures en quatre catégories :

**SISD** Parmi l'ensemble des architectures, on distingue tout d'abord les architectures intrinsèquement séquentielles disposant d'un unique flot d'instructions dans lequel chaque instruction intervient sur une unique donnée (scalaire). On parle dans ce cas de modèle d'exécution SISD (Single Instruction, Single Data).

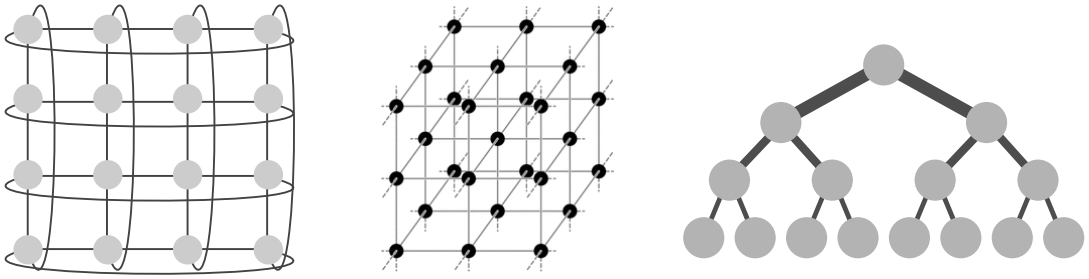
**SIMD** Viennent ensuite les architectures disposant d'un unique flot d'instructions manipulant plusieurs données à chaque instruction exécutée. On parle alors de modèle SIMD (Single Instruction, Multiple Data). Ces architectures sont particulièrement adaptées aux calculs vectoriels. L'architecture d'une majorité de processeurs graphiques dérive de ce modèle d'exécution.

**MISD** Le modèle d'exécution MISD (Multiple Instruction, Single Data) regroupe les architectures qui sont capables d'exécuter plusieurs traitements (instructions) sur une unique donnée. Les architectures à base de pipelines font partie de cette catégorie.

**MIMD** Enfin, le modèle MIMD (Multiple Instruction, Multiple Data) est l'un des plus fréquemment rencontré et adapté par la majorité des architectures parallèles actuelles. Ce type d'architecture comprend plusieurs unités d'exécutions (PEs) qui possèdent chacune un flot de données qui leur est propre et permettant à chacun de traiter des données différentes. Ce modèle englobe une grande variété d'architectures parallèles et il convient donc de les différencier en sous-catégories.

La classification de Duncan [10] précise la taxonomie de Flynn en divisant les architectures basées sur le modèle MIMD en deux groupes distincts : les machines à mémoire partagée et les machines à mémoire distribuée. Les machines à mémoire partagée détiennent un espace mémoire unique auquel toutes les unités de traitement peuvent accéder. Dans les machines à mémoires distribuées, chacune des unités de traitement dispose d'un espace mémoire distinct. Chaque PE a uniquement accès à sa propre mémoire et ne peut donc pas accéder directement à la mémoire des autres PEs. Les PEs sont reliés par un réseau d'interconnexion afin d'échanger des données entre eux. Les machines hybrides sont les machines qui associent mémoire partagée et mémoire distribuée. Elles sont généralement constituées d'un ensemble de machines à mémoire partagée (appelées nœuds) qui sont reliées entre elles via un réseau d'interconnexion.

On peut distinguer deux types de nœuds : les nœuds UMA (Uniform Memory Access) nommés aussi SMP (Symmetric multiprocessing) et les nœuds NUMA (Non Uniform Memory Access). Les nœuds SMP sont constituées de plusieurs PEs identiques connectés à une unique mémoire physique dont l'accès est uniforme d'un PE à un autre. Les nœuds NUMA disposent de plusieurs PEs connectés à plusieurs mémoires distinctes reliées entre elles par des mécanismes matériels. Ce type de nœud peut posséder bien plus de cœurs que les nœuds SMP en pratique (en contrepartie d'un temps d'accès mémoire généralement plus grand et variable d'un PE à un autre). Afin de couvrir la grande variabilité des architectures parallèles et la complexification des unités de calculs, de nouvelles classes d'architectures dérivées de la catégorie NUMA ont récemment vu



(a) Grille torique 2D (b) Grille torique 3D (c) Fat tree

FIGURE 3.1 – Quelques topologies réseau : les points représentent les PEs et les traits symbolise les connexions entre eux. L'épaisseur des traits correspond le débit des liens réseau.

le jour, telle que l'architecture hUMA (heterogeneous Uniform Memory Access) détaillant le fonctionnement des mécanismes de cohérence de cache mémoire dans les unités de calcul.

Les réseaux d'interconnexion peuvent être eux aussi classés en plusieurs groupes en fonction de la topologie qu'ils adoptent. Il existe une grande variété de topologies réseau, mais nous allons nous concentrer uniquement sur celles qui sont les plus utilisées dans les supercalculateurs [11] tels que les réseaux maillés, les fat tree, les grilles toriques 2D/3D. La structure des réseaux maillés peut être assimilée à un simple graphe quelconque ou chaque nœud du graphe représente un PE et chaque arc une connexion réseau directe entre deux PEs. Les topologies de grilles toriques 2D ou 3D (*cf.* figures 3.1a et 3.1b) sont assimilables à des grilles de machines (2D ou 3D) dont les extrémités ont été repliées sur elle même. La bande passante maximale est la même pour tous les liens dans cette de topologie réseau. La topologie fat tree (*cf.* figure 3.1c) correspond à un arbre dans lequel chaque nœud est une machine et chaque branche est une connexion réseau directe. La bande passante croit inversement proportionnellement en fonction de la distance entre la racine et les nœuds interconnectés.

On distingue les ressources de architectures parallèles de la manière de les organiser et de les rendre accessible. Nous allons nous intéresser aux techniques mis en œuvres pour gérer ces ressources et services présent sur les architectures parallèles.

## 3.2 Infrastructures parallèles

On peut diviser les infrastructures parallèles en trois grandes catégories : les grappes de serveurs, les grilles de calcul et les nuages.

**Grappes de serveurs** Encore appelés fermes de calcul, elles désignent un groupe de nœuds de calcul indépendants, homogènes (autant au niveau matériel que logiciel) et reliés entre eux par un réseau local rapide.

**Grilles de calcul** Elles regroupent une agglomération dynamique de ressources informatiques pouvant être délocalisées et qui tend à être plus hétérogène que la ferme de calcul. De plus, la coordination des ressources d'une grille est décentralisée contrairement aux grappes de serveurs. Les éléments constituant une grille communiquent entre eux en utilisant une grande diversité de réseaux (internet, réseaux privés, etc.).

**Les nuages** Ce concept est une extension de la notion de grille. Ce type d'infrastructure permet d'accéder aux ressources de manière plus transparente, depuis des appareils connectés à internet. De plus, ce type d'infrastructure considère les ressources comme extensibles et permet ainsi d'adapter leur utilisation en fonction des besoins.

Dans le but de répondre à des besoins divers tels que la maximisation des performances ou la minimisation de la consommation d'énergie, ces infrastructures peuvent mettre en place des techniques d'accélération matérielle, on parle alors d'accélérateurs. Ces outils visent à effectuer des calculs spécifiques à un domaine (traitement du signal, traitement graphique, etc.). Parmi les accélérateurs les plus utilisées, on peut trouver les GPU (Graphics Processing Unit). Initialement conçus pour accélérer les traitements graphiques, ils sont aujourd'hui aussi utilisés dans un cadre moins spécifique et visent à l'accélération de tout type de calcul vectoriel, on parle alors de GPGPU (General-Purpose Computing on Graphics Processing Units).

## 3.3 Paradigmes de programmation parallèles

Lorsqu'on programme sur des machines parallèles, plusieurs paradigmes s'offrent à nous. Parmi l'ensemble existant, nous allons nous restraindre uniquement à la mémoire partagée et au passage de messages pour concevoir des algorithmes de FFTs.

Dans le cas de la mémoire partagée, des zones mémoire sont accessibles et modifiables par plusieurs processus. On utilise alors des variables partagées comme vecteur de communication entre les processus. Il peut ainsi être utile de disposer de mécanismes de synchronisation pour qu'ils évitent de manipuler la même variable en même temps par exemple en utilisant des verrous, des sémaphores, ou même via de la mémoire transactionnelle. Ces mécanismes sont dans la majorité des cas implémentés matériellement, dans le système d'exploitation utilisé, ou dans des langages. On peut citer les threads POSIX [12] et OpenMP [13] comme technologies utilisant ce paradigme. Bien que le paradigme soit particulièrement bien adapté à la programmation sur machine à mémoire partagée, il est possible de l'utiliser sur les machine à mémoire distribuée via de la mémoire partagée distribuée (e.g. UPC [14]).

Dans le cas du passage de messages, les communications se font via des envois de messages entre les PEs. Les mécanismes de synchronisation tels que la barrière (permettant de synchroniser un certain nombre de processus à un point précis de leur exécution) effectuent une synchronisation via un passage de messages. On peut citer MPI [8] et Charm++ [15] comme technologies utilisant ce paradigme. Là aussi, bien que le paradigme soit particulièrement bien adapté à la programmation sur machine à mémoire distribuée, il est tout à fait possible de l'utiliser sur les architectures à mémoire partagée.

Sur les machines parallèles le mode de programmation le plus courant est le SPMD [16] (Single Program, Multiple Data), il consiste à exécuter, sur une architecture de type MIMD, un même de programme sur chacun des PEs. Chaque PE exécute les mêmes instructions d'un même programme sur des données différentes, à la manière des architectures SIMD, mais les multiples flots d'instructions peuvent potentiellement être désynchronisés entre eux. Des techniques de synchronisations sont prévus à cet effet. À contrario, le MPMD vise à exécuter des programmes différents sur chaque PE.

La spécialisation des calculs appliqués par les accélérateurs rend leur programmation différente de celle des processeurs centraux (CPU). On a ainsi vu naître de nouveaux langages tels que CUDA [17] dédié à la programmation des certains GPUs, ou de nouvelles interfaces de programmation (API) tels que OpenCL [18]. La démocratisation des accélérateurs a conduit à un besoin de simplifier leur programmation. C'est ainsi que le standard OpenACC [19] a vu le jour en proposant un moyen simple de paralléliser des calculs sur accélérateurs, à la manière

d'OpenMP. La dernière spécification d'OpenMP (version 4) a intégré de nouvelles directives visant à rendre plus transparente l'utilisation des accélérateurs. Malgré ces efforts, l'obtention de bonnes performances sur les accélérateurs reste encore trop souvent une tâche difficile nécessitant une bonne connaissance de leur fonctionnement.

## 3.4 Discussion

Les architectures parallèles présentent une grande variabilité et complexité. Par exemple, l'architecture du cluster de nœuds fins du supercalculateur Curie est composé d'un ensemble de nœuds NUMA interconnectés via un réseau InfiniBand disposant d'une topologie en fat tree ; chaque nœud dispose de quatre processeurs, qui contiennent eux-mêmes huit cœurs chacun ; chaque cœur dispose d'unité SIMD permettant de traiter plusieurs données simultanément par instruction. Le parallélisme sur ce supercalculateur s'effectue donc à plusieurs niveaux : au sein des cœurs, des processeurs, des nœuds et de l'ensemble du supercalculateur. La diversité des architectures matérielle est grande car elles peuvent différer celon le choix des processeurs, des disques, des accélérateurs, du type de réseau d'inteconnexion, etc. Afin de différencier les architectures et d'exprimer leur variabilité, des classifications présentées précedemment ont été misent en place.

La façon de programmer sur une architecture parallèle peut impacter les performances et peut présenter des limitation en ne proposant pas des mécanismes offert par le matériel. En effet, les choix des langages utilisés, les algorithmes implémentés, les optimisations appliquées, etc. peuvent dépendre de l'architecture matérielle sur laquelle ils reposent (*e.g.* utilisation de GPUs). Par exemple, l'utilisation du langage CUDA est plus adapté à la programmation GPU que l'utilisation d'UPC.

Nous allons nous intéresser au cas du calcul de FFT 3D car c'est une opération couramment appliquée par les applications scientifiques de haute performance et qu'il permet d'exposer assez simplement la relation qui unit l'architecture logicielle à l'architecture matérielle et donc l'intérêt d'adapter l'implémentation en fonction de cette dernière.

# 4 3D FFT

## 4.1 Aperçu

La transformée de Fourier est une opération mathématique permettant de décrire le l'ensemble des fréquences présentes dans d'un signal (fonction intégrable). Elle s'exprime comme une "somme infinie" de fonctions trigonométriques. Une telle sommation se présente sous forme d'intégrale. La transformée de Fourier $F$ d'une fonction $f$ sur $\mathbb{R}$ est donnée par l'expression 4.1. Cette transformation est réversible et il est donc possible retrouver la fonction $f$ à partir de $F$ via se qu'on appelle une transformée de Fourier inverse, définie par l'expression 4.2. La transformée de Fourier peut aussi s'appliquer sur des fonctions multidimensionnelles telles que les fonctions définies sur $\mathbb{R}^2$ ou $\mathbb{R}^3$.

$$F(\omega) = \int_{-\infty}^{+\infty} f(x) e^{-i\omega x} \, \mathrm{d}x \tag{4.1}$$

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} F(\omega) e^{+ix\omega} \, \mathrm{d}\omega \tag{4.2}$$

Cette transformation est utilisée dans de nombreux domaines tels que la dynamique moléculaire, médecine, astrophysique, météorologie, tomographie 3D [20], etc. Cependant, vu que les ordinateurs actuels peuvent effectuer bien plus efficacement des calcul discret bornés que des calculs continus non bornés, on utilise un l'équivalent discret de la transformée de Fourier dans les applications scientifiques : la transformée de Fourier discrète (DFT). La complexité en temps du calcul de la DFT est de $O(N^2)$ où N est la taille des données. Comme le calcul est relativement couteux en temps sur de grandes tailles de données, on a recours à une méthode plus efficace pour calculer des DFTs : les FFTs.

La transformée de Fourier rapide (FFT) est une classe d'algorithmes efficaces et très utilisée pour calculer la DFT. Par exemple, la FFT est utilisée par le logiciel de simulation de dynamique moléculaire GROMACS [21]. Elle est aussi utilisée pour résoudre efficacement des équations différentielles [22] ou encore pour accélérer la multiplication de grands nombres [23].

Ces algorithmes ont été implémentés dans de nombreuses bibliothèques de calcul numérique, qu'elle soit parallèle ou séquentielle.

## 4.2 FFTs séquentielles

L'algorithme de FFTs proposé par Cooley et Tukey en 1965 [24] est un algorithme célèbre et efficace permettant de calculer la DFT en temps $O(N\ log(N))$ (où N représente la taille de l'entrée). Du fait de son efficacité, beaucoup de bibliothèques de FFTs l'utilisent.

Les FFT multidimensionnelles peuvent être calculées en appliquant des FFT unidimensionnelles sur chaque dimension de la matrice de données d'entrée. Typiquement, en deux dimensions, cela revient à appliquer une FFT sur toutes les lignes de la matrice d'entrée, puis de réappliquer ensuite le traitement sur toutes les colonnes (cf. figure 4.1). L'ordre des dimensions traité n'a pas d'importance. On peut alors traiter les colonnes en premier, puis les lignes. Il en est de même en trois dimensions où chaque dimension est traitée l'une après l'autre en suivant la même méthode.

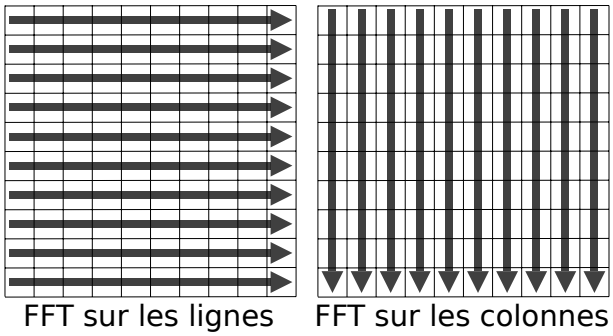



FIGURE 4.1 – Calcul de FFTs en deux dimensions sur une matrice

Cela peut être réalisé en utilisant plusieurs phases de calcul de FFTs unidimensionnelles appliquées le long d'un même axe, entrelacées par des phases de transposition qui réorganises les données, de façon à ce que les données le long de l'axe choisi n'aient pas été déjà traitées. Dans les implémentations séquentielles, ce processus est très coûteux lorsqu'il est appliqué sur de grandes matrices.

## 4.3 Parallélisation de FFTs 3D

Les FFT multidimensionnelles parallèle ont fait l'objet de nombreuses recherches afin de pouvoir traiter de grandes matrices. Nous allons nous concentrer sur les FFTs en trois dimensions, car il existe de nombreuses études qui proposent des optimisations spécifiques impactant grandement sur les performances et que la sélection des optimisations à utiliser qui maximisent les performances est un choix complexe.

Les méthodes existantes permettant de calculer des FFT 3D en parallèle peuvent être classifiées en deux grands groupes : celles qui utilisent une transposition globale de matrice et celles qui utilisent un schéma de communication binaire [1] (*cf.* figure 4.2). L'utilisation d'une transposition globale est connue pour mieux passer à l'échelle sur les supercalculateurs pétaflopiques [2] récents, même sur les supercalculateurs disposant d'une topologie réseau de type hypercube [3], où les schémas de communication binaire peuvent être faits nativement. Puisque les approches qui utilisent une transposition globale passe mieux à l'échelle, nous avons donc porté notre attention sur ce type d'approche.

1. Schéma de communication qui pour $2^p$ PEs applique $p$ phases de communication où la $i^{ème}$ phase fait communiquer le nœud $n$ avec le nœud $n \oplus 2^{p-i}$
2. Architectures permettant d'effectuer approximativement $10^{15}$ opérations flottantes par secondes
3. Topologie réseau formant une hypercube dans laquelle chaque sommet représente un nœud de calcul et chaque arête représente un lien physique entre deux nœuds

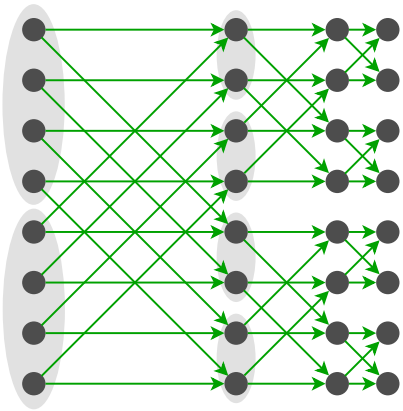


FIGURE 4.2 – Exemple de schéma de communication binaire : les points représentent les nœuds de calcul et les flèches représentent les communications

Introduisons quelques notations et conventions relatives à la transposition globale. Considérons un cube de données de taille $N \times N \times N$ le long d'axes X, Y et Z. En mémoire, les cases le long de l'axe X sont organisées de manière contiguë. Ces cases forment des lignes qui sont elles-mêmes organisées le long de l'axe Y. Ces lignes formes à leur tour des blocs stockés de manière contiguë le long de l'axe Z.

Une manière de calculer une FFT 3D en utilisant une transposition globale est de distribuer des tranches de données (*slabs*) le long de l'axe Z entre les PEs et d'entrelacer des phases de calcul et de transposition. Ainsi, chaque PE stocke un bloc de taille $N \times N \times n_z$ où $n_z \leq N$. L'algorithme 1 et la figure 4.3 fournissent un aperçu de cette première approche. Cette méthode se nomme décomposition en pinceaux (*slab*) ou décomposition 1D [21] car les données sont distribuées selon une seule dimension (dans ce cas l'axe Z). La transposition du cube de données peut se faire en utilisant un échange total, aussi connu sous le nom All-to-All (*e.g.* en utilisant les collectives MPI tel que MPI_Alltoall ou MPI_Alltoallv) suivi par une transposition locale. Cette approche est efficace tant que le nombre de PEs ne surpasse pas $N$. En effet, lorsque $N$ PEs sont utilisés, chacun d'eux manipule une tranche de hauteur 1 et il est impossible de distribuer les données sur plus de $N$ PE avec cette méthode.

**Algorithme 1 :** Schéma de décomposition 1D

**Données** : Tranches de données sur les axes X et Y dans le domaine spatiale
**Résultat** : Tranches de données sur les axes X et Y dans le domaine fréquentiel
1 Appliquer des FFTs 2D sur chaque tranches de données locales;
2 Transposition XZ des tranches de données;
3 Appliquer des FFTs 1D sur l'axe $X$ chaque tranches de données locales;
4 Transposition XZ des tranches de données;

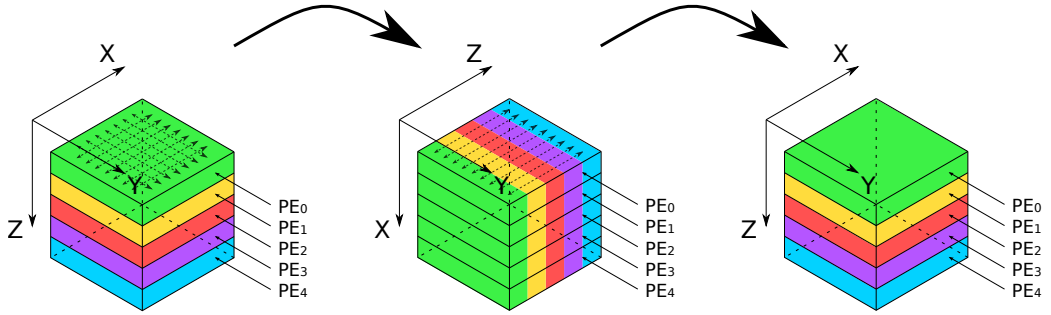



FIGURE 4.3 – Schéma de décomposition 1D avec une transposition XZ. Les couleurs représentent l'organisation des data et les flèches en pointillés représentent les phases de calcul de FFT.

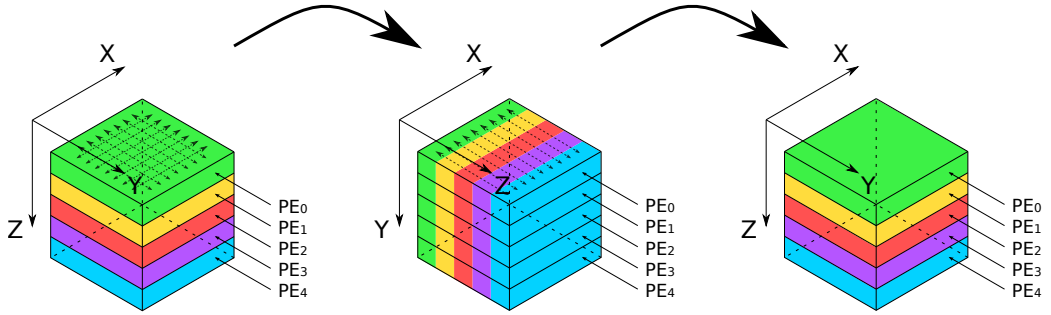



FIGURE 4.4 – Schéma de décomposition 1D avec une transposition YZ.

Notez qu'il est possible d'utiliser soit une transposition XZ (montré sur la figure 4.3), soit une transposition YZ (montré sur la figure 4.4). En fonction de la transposition qui est utilisée, le schéma d'accès mémoire diffère et peut ainsi avoir un impact sur les performances qui dépendent elles-mêmes de l'architecture matérielle utilisée.

La limite imposée par la décomposition 1D en terme de passage à l'échelle pose problème sur les supercalculateurs récents qui peuvent posséder des millions de cœurs [11]. Pour surpasser cette limitation, les données peuvent être distribuées le long de deux axes : Y et Z. Ainsi, au lieu d'être distribuées en tranches, les données sont distribuées en pinceaux entre les PEs. On appelle donc cette méthode décomposition en pinceaux (pencil) ou décomposition 2D [21]. Un aperçu de cette méthode est donné par l'algorithme 2 et la figure 4.5. Notez que les transpositions XY et XZ peuvent être échangées dans cet algorithme. Cette approche permet d'utiliser jusqu'à $N^2$ PEs (alors que la décomposition 1D limite à seulement $N$ PEs).

Notez que même si la décomposition 2D passe bien mieux à l'échelle que son homologue 1D, pour un même nombre de cœurs et si la décomposition 1D le permet, les performances de la décomposition 1D sont généralement meilleures que celles d'une décomposition 2D, car une transposition supplémentaire est nécessaire dans le second cas [20].

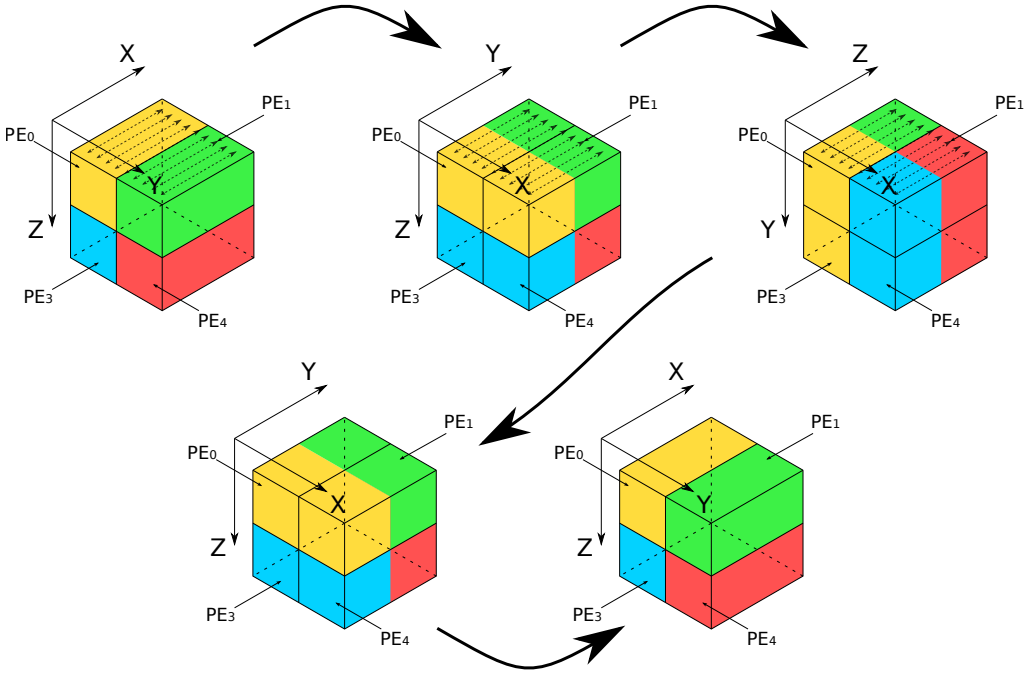



FIGURE 4.5 – Schéma de décomposition 2D.

**Algorithme 2 :** Schéma de décomposition 2D

**Données** : Pinceau de données le long de l'axe X dans le domaine spatial
**Résultat** : Pinceau de données le long de l'axe X dans le domaine fréquentiel

1 Appliquer des FFTs 1D sur chaque pinceaux de données sur l'axe X;
2 Transposition XY des pinceaux de données;
3 Appliquer des FFTs 1D sur chaque pinceaux de données sur l'axe X;
4 Transposition XZ des pinceaux de données;
5 Appliquer des FFTs 1D sur chaque pinceaux de données sur l'axe X;
6 Transposition XZ des pinceaux de données;
7 Transposition XY des pinceaux de données;

## 4.4 Optimisations de FFTs 3D

Au-delà du choix de la décomposition 1D ou 2D étudié plus haut, de multiples optimisations peuvent être appliquées sur l'algorithme de FFTs initial afin de tirer profit de la diversité des architectures matérielles utilisées ou des paramètres spécifiques de l'application.

Tout d'abord, il est possible d'optimiser l'échange total effectué par l'algorithme. En effet, plusieurs algorithmes existent et leurs performances dépendent de plusieurs paramètres tels que la taille des messages, la topologie réseau, la latence des liens du réseau, etc. Par exemple, Bruck et al. ont proposé un algorithme qui est particulièrement efficace lorsque la taille des messages est petite [25] ; Prisacari et al. ont proposé un algorithme qui est spécifiquement efficace pour des messages de grande taille dans les réseaux d'interconnexion de type fat tree [26]. Les implémentations MPI telles qu'OpenMPI et MPICH sélectionnent ces algorithmes au lancement ou à l'exécution pour tenter de maximiser les performances [27, 28].

De plus, sur certaines architectures matérielles, la distribution des données peut avoir un impact significatif sur les performances. Par exemple, l'opération collective MPI MPI_Alltoallv, qui est utilisée pour effectuer un échange total avec des tailles de données variant entre les PEs

est peu efficace sur les supercalculateurs Cray XT [20]. Pour pallier ce problème, les tampons mémoires d'envois et de réceptions sont agrandis de telle manière à ce qu'ils aient tous la même taille et la collective MPI MPI_Alltoall est utilisée en remplacement. En effet, cette collective ne manipule que de données équitablement distribuées entre les PEs. Cette opération permet d'améliorer les performances sur les supercalculateurs Cray XT. De manière similaire, cette famille de supercalculateurs met en place un mécanisme de mémoire partagée entre les nœuds qui peut être utilisée pour accroître les performances des applications qui l'utilisent [29].

Un autre type d'optimisation possible est de recouvrir le temps de calcul avec le temps de communication. Cette optimisation peut introduire un surcoût supplémentaire, modifier les schémas de communication et donc na va pas nécessairement améliorer les performances de l'algorithme. Kandalla et al. [30] ont montré que l'on peut effectuer un recouvrement quasi-parfait sur les FFT 3D en utilisant une technique de déchargement réseau. Cette technique permet d'améliorer les performances d'au maximum 23% par rapport à une version ne disposant pas de recouvrement en utilisant InfiniBand.

Enfin, certaines optimisations tirent profit des contraintes fixées par l'application qui applique des FFTs. Examinons, par exemple, le cas d'une application qui appliquerait un produit de convolution sur un cube de données. Cette opération requiert deux éléments : le cube de données d'entrée et un filtre de convolution (paramétrant le comportement du produit de convolution). Un moyen efficace d'appliquer cette opération est d'appliquer une FFT sur le cube de données et une autre sur le filtre de convolution, puis, multiplier les valeurs résultantes entre elles et appliquer une FFT inverse sur le produit des deux cubes [31]. Dans ce cas, deux transpositions peuvent être évitées durant tout le traitement si une décomposition 1D est utilisée [20]. En effet, la dernière transposition effectuée par l'algorithme de décomposition 1D sert uniquement à replacer la matrice dans sa configuration initiale. Ce n'est pas nécessaire ici, car la multiplication des cubes de données entre eux est possible quelle que soit l'organisation des matrices traitées, pourvu que ces matrices aient la même organisation.

## 4.5 Bibliothèques de FFTs 3D

### Bibliothèques séquentielles

Un grand nombre de bibliothèques, open-source ou commerciales, fournissent des implémentations de FFTs séquentielles. La bibliothèque FFTW [2] (Fastest Fourier Transform in the West), développée par Matteo Frigo et Steven G. Johnson, est l'une des bibliothèques multi-plateformes les plus utilisées. Cette bibliothèque est conçue de telle manière à être performante sur de nombreuses architectures matérielles. D'autres bibliothèques réputées et open-source permettent de calculer des FFTs. C'est le cas d'Eigen [32] et la GSL [33] (GNU Scientific Library). On peut aussi citer le projet SPIRAL dont les objectifs et la méthode sont similaires à la FFTW, mais qui se veut moins spécifique, s'étendant ainsi au cas des algorithmes de traitement du signal. Parmi les alternatives commerciales existantes, on peut citer la ESSL (Engineering and Scientific Software Library) d'IBM et la MKL (Math Kernel Library) d'Intel.

### Bibliothèques parallèles

Certaines bibliothèques fournissent des implémentations multithreads et dans certains cas des implémentations distribuées qui sont généralement basées sur MPI. Beaucoup de ces bibliothèques utilisent des implémentations séquentielles reconnues afin de pouvoir fournir une version parallèle. Cela vient du fait qu'un unique calcul de FFTs est décomposable en plusieurs petits calculs de FFTs indépendants [24], pouvant être effectués en parallèle. Ainsi, les bibliothèques

telles que la FFTW ou la MKL, mentionnées avant, fournissent leur propres implémentations parallèles et d'autres bibliothèques comme P3DFFT [20] (Parallel Three-Dimensional Fast Fourier Transforms) et 2DECOMP&FFT [29] sont basées sur des implémentations séquentielles de FFTs externes, tel que FFTW et ESSL. Les bibliothèques FFTW et 2DECOMP&FFT ont été considéré comme référence pour les expériences réalisées dans ce rapport.

**FFTW** Plusieurs variantes de la FFTW (version 3) existent : une implémentation utilisant des threads (qui n'en utilise qu'un seul par défaut), une version MPI dédiée aux architectures à mémoire distribuée (proposant uniquement une décomposition 1D) et une implémentation écrite en Cilk dédiée aux architectures centralisées à mémoire partagée (SMP). Afin d'être performante, la FFTW utilise des stratégies d'auto-optimisation : des morceaux de codes C hautement optimisés nommés codelets conçus pour calculer des FFTs sur des données de petite taille sont assemblés ensemble pour former un programme durant une phase de planification à l'exécution. Les codelets peuvent être automatiquement créés en utilisant un générateur nommé genfft nécessitant une description mathématique de haut niveau d'un algorithme de DFT. La version parallèle multithread de la FFTW est une variante spécifique qui supporte des planifications parallèles.

**SPIRAL** Mis au point dans les années 2000 et toujours activement développé, le projet SPIRAL [34] a pour but de générer et d'optimiser automatiquement des algorithmes de traitement du signal, incluant ainsi, les algorithmes de DFT. Tout comme la FFTW, SPIRAL utilise une approche mathématique qui permet de compiler des formules écrites dans un langage mathématique de haut niveau nommé SPL en un code C. Les descriptions SPL de haut niveau peuvent être elles-mêmes automatiquement générées par un générateur de formules SPL. Tout comme la FFTW, ce projet propose une implémentation dédiée aux architectures à mémoire distribuée se basant sur MPI en plus d'une implémentation séquentielle [35].

**P3DFFT** Développé par Dmitry Pekurovsky, P3DFFT [20] est une bibliothèque open-source développée en utilisant le langage FORTRAN et permettant de calculer des FFT en parallèle sur des architectures à mémoire distribuée. Cette bibliothèque propose les décompositions 1D et 2D et a été mise au point pour passer à l'échelle sur des plate-formes pétaflopiques. P3DFFT peut utiliser plusieurs bibliothèques séquentielles telles que ESSL ou FFTW. Cette bibliothèque supporte les transformations de nombres réels vers des nombres complexes et inversement, mais la transformation de nombres complexes vers des nombres complexes est en cours d'implémentation et n'est pas disponible publiquement pour le moment. Afin de maximiser ses performances, P3DFFT implémente des optimisations spécifiques comme par exemple le remplacement de la collective MPI MPI_Alltoallv par MPI_Alltoall pour la famille de supercalculateurs Cray XT. Ces optimisations sont implémentées via la compilation conditionnelle. Selon l'auteur, P3DFFT passe à l'échelle jusqu'à au moins 65536 cœurs.

**2DECOMP&FFT** Développé par Ning Li et Sylvain Laizet, 2DECOMP&FFT [29] est une autre bibliothèque open-source programmée en utilisant le langage FORTRAN et permettant de calculer des FFT en parallèle sur des architectures à mémoire distribuée. Mais contrairement à P3DFFT, 2DECOMP&FFT supporte en plus les transformations de nombres complexes en nombres complexes. Tout comme P3DFFT, des optimisations spécifiques ont été implémentées (*e.g.* supercalculateurs Cray XT) via la compilation conditionnelle.

## 4.6 Discussion

Les architectures des supercalculateurs évoluant très vite, de nouvelles optimisations doivent être implémentés pour maximiser les performances sur divers types d'architectures matérielles. Chaque optimisation peut introduire du code supplémentaire et peut potentiellement impacter toute la structure de l'application. L'utilisation de la compilation conditionnelle conduit à l'entrecroisement de petit morceaux de codes spécifiquement dédiés à l'optimisation de l'application. Cela peut réduire significativement la lisibilité du code et surtout causer des problèmes de maintenance applicatif.

En pratique, aucune bibliothèque n'implémente toutes les optimisations. Or, les utilisateurs de des bibliothèques peuvent avoir besoin d'un groupe spécifique d'optimisations qui n'est pas toujours supporté par les bibliothèques existantes. Ils doivent donc soit modifier les bibliothèques existantes, soit implémenter leur propre code de FFTs spécifique.

Afin de réutiliser un maximum le travail fait sur les implémentations séquentielles de FFTs et leurs optimisations, beaucoup d'implémentations parallèles utilisent des implémentations séquentielles. Malheureusement, l'adaptation de ces bibliothèques parallèles à de nouvelles bibliothèques séquentielles ou à de nouvelles fonctionnalités qu'elles offrent est une tâche ardue, car leur utilisation est implémentée dans le cœur même des bibliothèques parallèles.

Une solution prometteuse pour manipuler facilement la structure des algorithmes de FFTs et leur spécialisation consiste à utiliser des modèles de composants.

# 5 Modèles de composants

## 5.1 Aperçu

**Définitions** Le concept de composants logiciels a été initialement proposé par Douglas McIlroy [36] en 1968 et a depuis fait l'objet de nombreuses recherches. La programmation orientée composants est une branche du génie logiciel visant à utiliser une approche modulaire au niveau de l'architecture des applications en assemblant d'unités logicielles appelées composants pour former un programme. Un modèle de composants définit ce qu'est un composant et comment les composer. On peut classer les modèles de composants en deux catégories : ceux qui effectuent une composition spatiale et ceux qui effectuent composition temporelle (*i.e.* workflow). Nous allons nous concentrer sur le premier groupe.

Il n'existe pas de définition communément admise de ce qu'est un composant. Néanmoins, une définition qui fait à peu près consensus est proposée par Clemens Szyperski : "Un composant logiciel est une unité de composition dotée d'interfaces spécifiées. Un composant logiciel peut être déployé indépendamment et être composé par des parties tierces." [4]. Dans de nombreux modèles de composants, les composants sont des boîtes noires réutilisables disposant de points d'interactions nommés ports (interfaces des composants) qui ont un nom (son rôle) et une description. La plupart des modèles de composants assurent la séparation des interfaces d'un composant de leurs implémentations permettant ainsi d'accroître la modularité de l'application.

Afin de former une application entière, les composants doivent être instanciés, puis assemblés. L'assemblage peut être réalisé en interconnectant les interfaces des composants entre elles. La nature des connexions est définie par le modèle de composants et varie beaucoup d'un modèle à un autre. Une fois assemblés, les composants peuvent communiquer entre eux via leurs interfaces en suivant un protocole fixé par le modèle.

**Avantages** Les modèles de composants visent à isoler au sein d'un composant les problématiques qu'il doit traiter (séparation des préoccupations) et à réutiliser des parties tierces applicatives. La séparation des préoccupations est possible en distinguant d'une part, la programmation des composants traitant des détails d'implémentation (bas niveau) et de l'autre, l'assemblage des composants gérant la structure même de l'application (haut niveau). La réutilisation de composants est possible car l'utilisation de composant nécessite uniquement l'accès à ses interfaces. Ainsi, il n'est pas nécessaire de connaître implémentation d'un composant pour pouvoir l'utiliser. La séparation des préoccupations et la réutilisation de composants nous permettent de facilement combiner des codes de FFTs de sources variées et de les assembler pour former des assemblages spécialisés calculant des FFTs. De fait, il ne devrait pas être nécessaire comprendre les détails d'implémentations ou de redévelopper des optimisations existantes afin d'adapter des FFTs.

## 5.2 Caractéristiques

Bien que la diversité des modèles de composants soit grande, il existe des particularités communes à de nombreux modèles. Nous allons tenter de décrire uniquement celles qui nous serons les plus utiles dans le cadre de ce stage. À savoir la hiérarchie, les connecteurs et la généricité des composants.

**Hiérarchie** La notion de hiérarchie [37] consiste à permettre de créer des composants dont l'instanciation est un assemblages d'autres composants. On parle alors de composants composites. Lorsqu'un composant n'est pas composite, on parle de composants primitifs. Dans Fractal [38], les composites ont un rôle plus actif puisqu'ils intègrent une notion de membrane qui permet d'intercepter les appels de méthodes sur leurs ports avant de les retransmettre à leurs instances internes. Cette hiérarchie joue un rôle déterminant dans la modularité des applications en permettant de voir les différents niveaux d'abstraction applicatifs.

**Connecteurs** Les connecteurs [39] sont des éléments permettant de relier des ports entre eux. Leur but est d'abstraire l'assemblage des composants. Ils permettent de définir la politique de communication entre les composants. Leur instanciation diverge en fonction du type de connecteur appliqué. Une fois instancié, un connecteur peut correspondre par exemple à un appel de méthode ou à du passage de messages.

**Généricité** La généricité [40] des composants s'apparente à la généricité dans les langages comme C++ ou Java (mécanisme de templates). Les composants génériques possèdent des paramètres génériques. Il est possible de réaliser une spécialisation de ces composants pour spécifier leur comportement lorsque des instances précises de paramètres sont fixées. Il est possible de réaliser des spécialisations soit manuellement, soit automatiquement via un algorithme. Pour instancier un composant générique, tous ces paramètres génériques doivent être fixés. Cette notion peut nous permettre de sélectionner certaines optimisations (en utilisant la spécialisation de composants) en fonction de critères applicatifs (*i.e.* paramètres génériques).

## 5.3 Modèles distribués

Dans le cadre de ce stage, on s'intéresse uniquement aux modèles de composants distribués. Parmi eux, on peut citer notamment CCM [41] (CORBA Component Model) et GCM [42] (Grid Component Model). Malheureusement, ces deux modèles introduisent une charge de travail supplémentaire lors de l'exécution qui est satisfaisante pour la plupart des applications, mais inacceptable pour du calcul de haute performance [43].

CCA [44] (Common Component Architecture) est une initiative du département de l'énergie des États-Unis afin de mettre en avant l'utilisation des composants dans le domaine de la haute performance. Dans CCA, les composants sont uniquement décrits par un ensemble de fonctions. Leur description se fait via un langage nommé SIDL (Scientific Interface Definition Language). Les composants détiennent une collection de ports dynamique (pouvant être définie et modifiée à l'exécution) et chaque port peut accéder à un sous-ensemble des fonctions du composant auquel il est affilié. CCA propose deux types de ports : les ports *provide* permettant de fournir des services à d'autres composants et les ports *use* permettant d'utiliser les services d'autres composants.

Le parallélisme est introduit dans le modèle via des composants parallèles. Une instance d'un composant parallèle est formée d'un ensemble de processus appelé cohorte. Chaque processus d'une cohorte peut alors traiter des données en parallèle. On parle d'approche SCMD (Single Component Multiple Data). CCA permet aussi d'exécuter simultanément plusieurs cohortes (sur des PEs disjoint) provenant d'instance de composant différents. On parle dans ce cas d'approche MCMD (Multiple Component Multiple Data). Sachant que la taille des cohortes peut varier d'une instance à une autre, il est nécessaire de définir comment coupler les cohortes. Ce problème, connu sous le nom de couplage $M \times N$, est résolu en utilisant un composant de couplage qui va effectuer une redistribution des données. Notez que CCA est un modèle de composants dynamique qui permet de décrire des ports différents dans chaque processus de la cohorte.

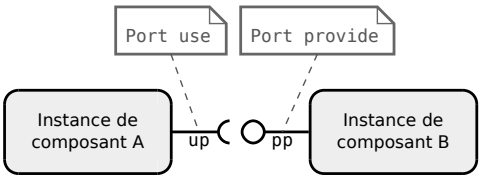



FIGURE 5.1 – Deux instances de composants CCA connectées par leurs interfaces.

Après instanciation, les instances de composants sont assemblées en connectant les ports *use* aux ports provide qui exposent une description compatible. La communication entre les instances de composants se fait via des appels de procédures sur les ports. La communication au sein d'une cohorte n'est pas assurée par le modèle de composant. Ce rôle est relayé à une bibliothèque de passage de messages telle que MPI ou PVM. CCA permet de reconfigurer l'assemblage lors de l'exécution permettant ainsi d'ajouter ou de supprimer des instances ou des connexions à la volée. La figure 5.1 donne un exemple d'un assemblage contenant deux instances connectées entre elles par une connexion use-provide.

Le concept de cohorte proposé par CCA limite les processus à ne contenir qu'un seul composant ce qui peut être un frein à la construction d'application à grains fins. De plus, on attend d'un modèle de composants d'exposer toutes les interactions possibles entre les composants dans l'assemblage. Or les communications internes à une cohorte n'apparaissent pas dans CCA. Cela peut rendre plus difficile la vérification des applications et peut constituer un frein à leur maintenabilité.

Le modèle L$^2$C [7] (Low Level Component) est un modèle HPC minimaliste se basant sur C++ et FORTRAN n'introduisant aucune charge de travail supplémentaire à l'exécution. Il fournit des composants primitifs, des connexions locales (port use-provide C++ et FORTRAN), des connexions MPI (partage de communicateurs MPI) et des connexions CORBA. Avant d'étudier la capacité du modèle L$^2$C à pouvoir exprimer des algorithmes de FFTs 3D ainsi que leur adaptation, nous allons tout d'abord détailler un peu le modèle.

## 5.4 L$^2$C

Le modèle L$^2$C peut être vu comme à la fois comme une extension du processus de compilation modulaire et comme un modèle de composants de bas niveau proche du système. En effet, chaque composant est compilé en un fichier objet (plus précisément en une bibliothèque dynamique dans les implémentations actuelles). Au lancement, les composants sont instanciés et les instances sont connectés en suivant les directives présente dans un fichier de description d'assemblage (LAD).

L$^2$C supporte de nombreuses fonctionnalités telles que le partage de la mémoire, l'appel de procédures (native) C++ ou FORTRAN, le passage de message via MPI, l'appel de procédure à distance via CORBA. Tout comme CCA, les composants L$^2$C fournissent des services via des ports *provide* et en utilise d'autres via des ports *use* et la communication entre les instances de composants se fait aussi via des appels de procédure sur les ports. Les services doivent être déclarés en tant qu'interface C++, FORTRAN ou CORBA. Plusieurs ports *use* peuvent être connectés à un même port *provide*, on parle alors de ports *use multiple*. Un port est associé à une interface objet et un nom. L$^2$C fourni aussi des ports MPI qui permettent aux instances de composants de partager des communicateurs MPI. Les composants peuvent aussi exposer des attributs utilisés pour configurer les instances. Dans l'implémentation C++, les composants L$^2$C sont des classes C++ auxquelles sont ajoutées des annotations nécessaires à la déclaration de composant (nom, ports et propriétés).

Un assemblage L$^2$C peut être décrit en utilisant un fichier de description d'assemblage. Ce fichier contient la description de toutes les instances de composant, les valeurs de leurs attributs et les connexions entre les instances. Chaque instance de composant est attachée à un processus et à chaque processus est associé à un point d'entrée (une interface qui est appelée lorsque l'application démarre). Ce fichier contient aussi la configuration des ports MPI.

L$^2$C fourni aussi une API [1] C++/FORTRAN pour instancier, détruire, configurer et connecter les instances de composants.

La description d'assemblage de L$^2$C a besoin d'être réécrite pour chaque architectures matérielle ciblée ou à chaque modification des paramètres applicatifs. Comme ce processus est délicat et est sujet à des erreurs, de telles descriptions devraient être automatiquement générées. Un modèle de composants de plus haut niveau automatisant la génération de l'assemblage peut accroître la maintenabilité de la génération d'assemblage et faciliter le développement de nouveaux assemblages. C'est l'une des intentions de HLCM.

## 5.5 HLCM

HLCM [5] (High Level Component Model) est un modèle de composants proposant un haut niveau d'abstraction dédié au calcul à haute performance. C'est un modèle hiérarchique, générique et basé sur des connecteurs. Le but d'HLCM est de produire, à partir d'un assemblage de haut niveau, un assemblage de bas niveau exprimé dans un autre modèle de composants interchangeable plus proche du système tel que L$^2$C ou Gluon++ (basé sur le langage Charm++ [15]). L$^2$C et Gluon++ étant non hiérarchiques, non génériques et n'étant pas basés sur des connecteurs, HCLM doit exprimer particularité provenant modèle de composants de haut niveau avec celles qui sont présentes dans le modèle sous-jacent utilisé lors de la transformation de l'assemblage. L'assemblage de haut niveau ne peut pas être modifié à l'exécution (car il est compilé). Néanmoins, cette technique permet de profiter des avantages offert par la hiérarchie, les connecteurs et la généricité sans introduire une charge de travail supplémentaire à l'exécution par rapport au modèle sous-jacent. Il convient alors d'utiliser un modèle sous-jacent efficace pour produire des applications HPC performantes. L$^2$C est un bon candidat pour cela, car il n'introduit pas de charge de travail supplémentaire à l'exécution. On peut ainsi supposer que HLCM est un modèle de choix pour programmer des composants logiciels sur des architectures distribuées de haute performance.

## 5.6 Discussion

Les modèles de composants permettent de manipuler facilement la structure des applications en assemblant des unités logicielles nommées composants. La séparation des préoccupations et la réutilisation sont deux concepts apportés par cette approche qui peuvent nous permettre de combiner et assembler facilement des codes de FFTs pour former des assemblages spécialisés.

Parmi les modèles de composants proposant un modèle d'exécution assez performant pour pouvoir implémenter des applications HPC, on trouve CCA et L$^2$C. Étant donnée que L$^2$C rend explicite les interfaces MPI contrairement à CCA et qu'il permet aussi de manipuler la structure des applications à un grain plus fin que CCA (notamment au niveau de la composition interne aux processus), nous l'avons sélectionné pour concevoir des algorithmes de FFT 3D.

1. interface de programmation

Les concepts de hiérarchie, de généricité et de connecteurs présents dans HLCM nous permettraient de décrire les applications de FFT avec un haut niveau d'abstraction, puis via une transformation de l'assemblage, de générer automatiquement un assemblage $L^2C$ spécialisé, dans le but d'adapter les performances en fonction du contexte d'exécution.

# 6 Implémentation d'algorithmes de FFTs 3D en L$^2$C

## 6.1 Aperçu

Ce chapitre vise à analyser comment L$^2$C peut être utilisé pour écrire des assemblages de FFT 3D basés sur l'utilisation de transpositions globales (voir Section 4.3). Nous avons dans un premier temps conçu un assemblage de FFT 3D élémentaire. Puis nous l'avons modifié pour prendre en compte quelques optimisations présentées en Section 4.4. Les optimisations ont été appliquées en trois étapes afin de mettre en avant les possibilités offertes par le modèle de composants :

i) en remplaçant l'implémentation de composant avec une implémentation plus performante sur certaines architectures (Section 6.3)

ii) en utilisant les attribut des composants afin de pendre en compte les plate-formes hétérogènes (Section 6.4)

iii) en modifiant l'assemblage lui-même pour implémenter un recouvrement entre les calculs et les communications ou une décomposition 2D (Section 6.5).

Tous les assemblages présentés ici ont été implémentés en C++ en utilisant L$^2$C et certains sont évalués dans le chapitre 7.

## 6.2 Assemblage de base

### Assemblage local

La figure 6.1 montre un assemblage qui implémente l'algorithme 1. Dans cet algorithme, on peut identifier 8 tâches distinctes qui peuvent être encapsulées dans des composants : 6 tâches de calcul et 2 tâches de contrôle. Les composants dédiés au calcul implémentent les tâches suivantes :

1. `Allocator` : alloue des tampons mémoire 3D.
2. `SlabDataInitializer` : initialise les données d'entrées.
3. `Planifier1D_X` et `Planifier2D_XY` : planifie des FFTs séquentielles efficaces.
4. `FFTW` : calcul des FFTs (encapsule la bibliothèque FFTW).
5. `MpiTransposeSync_XZ` : effectue une transposition globale des données.
6. `SlabDataFinalizer` : finalise les données de sortie (stockage, vérification ou réutilisation).

Les autres composants de contrôle implémentent les tâches suivantes :

7. `ProcessingUnit` : regroupe les calculs de FFT et les transpositions au sein d'une unique unité de traitement (diffuse les ordres reçus sur ses ports provide vers ses ports use).
8. `SlabMaster` : contrôle la structure globale de l'application (gère les tampons de données sur lesquels les composants travaillent, déclenche la phase d'initialisation, de calculs, et de finalisation).

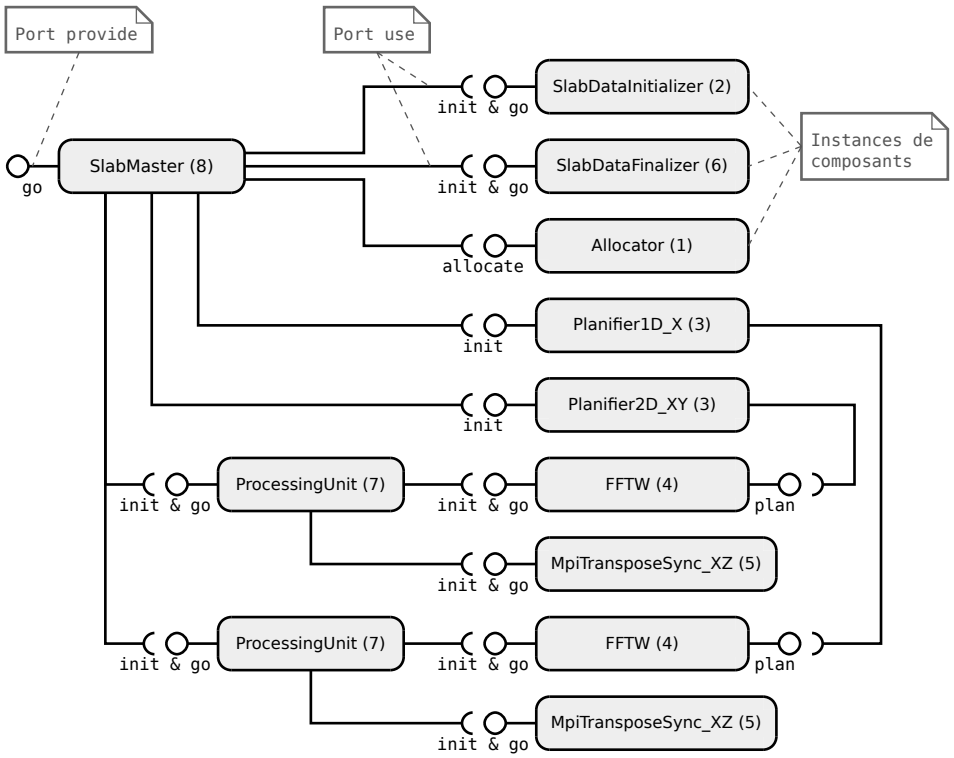



FIGURE 6.1 – Assemblage élémentaire de FFT 3D local à un processus utilisant une décomposition 1D.

Les tâches 2 (`SlabDataInitializer`), 6 (`SlabDataFinalizer`) et 8 (`SlabMaster`) sont spécifiques à la décomposition 1D. Ces tâches, en plus de la tâche 5 (`MpiTransposeSync_XZ`) sont spécifiques à une stratégie de parallélisme donnée (*i.e.* passage de messages via MPI). La tâche 3 (`Planifier1D_X` et `Planifier2D_XY`) et la tâche 4 (`FFTW`) est propre à la bibliothèque choisie pour calculer des FFT séquentiellement.

Tous les composants dédiés au calcul à l'exception du composant `Allocator` se basent sur des tampons mémoires pour le traitement des données. Il peuvent en utiliser soit deux (*i.e.* un tampon d'entré et un tampon sortie) si le calcul qu'il effectue n'est pas fait sur-place, soit qu'un seul si le calcul qu'il effectue est fait sur-place. Les tampons sont initialisés via un passage d'adresses mémoires (pointeurs C++) au démarrage de l'application. Pour cela, ces composants fournissent un port `Init` visant à fixer les adresses mémoires des tampons, mais aussi à initialiser ou libérer des ressources temporaires (*e.g.* tampons MPI). Lorsqu'un seul tampon est nécessaire, l'adresse du tampon d'entrée et de sortie sont les mêmes. Ces composants fournissent aussi un port `Go` permettant de lancer le calcul dont il sont responsables en récupérant les données du tampon d'entrée (fixé lors de l'initialisation) et écrivant les nouvelles dans le tampon de sortie (fixé aussi lors de l'initialisation).

Étant donnée que le prototype de la fonction de planification de FFTs dépend de la bibliothèque de FFT choisie, le composant FFT (tâche 4) expose une interface spécifique nommée `Plan` utilisé par `Planifier1D_X` et `Planifier2D_XY`. Cette connexion est utilisé pour configurer les composants `FFT` après la phase de planification.

L'application fonctionne en trois étapes. La première consiste à initialiser toute l'application en allouant tous les tampons, puis en diffusant leur adresses mémoires entre les instances de composants, en planifiant ensuite les FFTs et enfin en diffusant les plans. La seconde étape consiste à appliquer les calculs sur les données en appelant une méthode sur le port `Go` des instances afin d'entrecroiser le calcul de FFT et les communications. L'étape finale consiste à libérer toutes

ressources occupées comme les tampons mémoires. Toutes ces étapes sont déclenchées par un appel de méthode sur le port `Go` de l'instance du composant `SlabMaster`.

L'assemblage à été conçu de telle manière à être configuré pour un calcul spécifique de FFT 3D. La taille des tampons les déplacements mémoires sont décrit en temps qu'attributs et ne sont pas calculés à l'exécution. Cela permet de minimiser le surcoût associé à leur gestion.

### Assemblage distribué

La figure 6.2 présente une extension de l'assemblage précédent. Cette nouvelle version est obtenue en remplaçant, dans un premier temps, quatre composants par des versions qui disposent d'un port MPI. Ces quatre composants sont : `SlabDataInitializer` (tâche 2), `MpiTransposeSync_XZ` (tâche 5), `SlabDataFinalizer` (tâche 6) et `SlabMaster` (tâche 8). Puis, dans un second temps, en répliquant l'assemblage local sur chaque processus MPI. Les instances du composant MpiTransposeSync_XZ d'une même phase de calcul sont connectées entre elles par leur port MPI, afin qu'elles partagent un communicateur MPI. Il en est de même pour les instances des composants `SlabMaster`, `SlabInitializer` et `SlabFinalizer`. Cet assemblage porte le nom de version `L2C_1D_2t_xz` et est utilisé comme assemblage de base pour évaluer la réutilisation de l'application au chapitre 7.

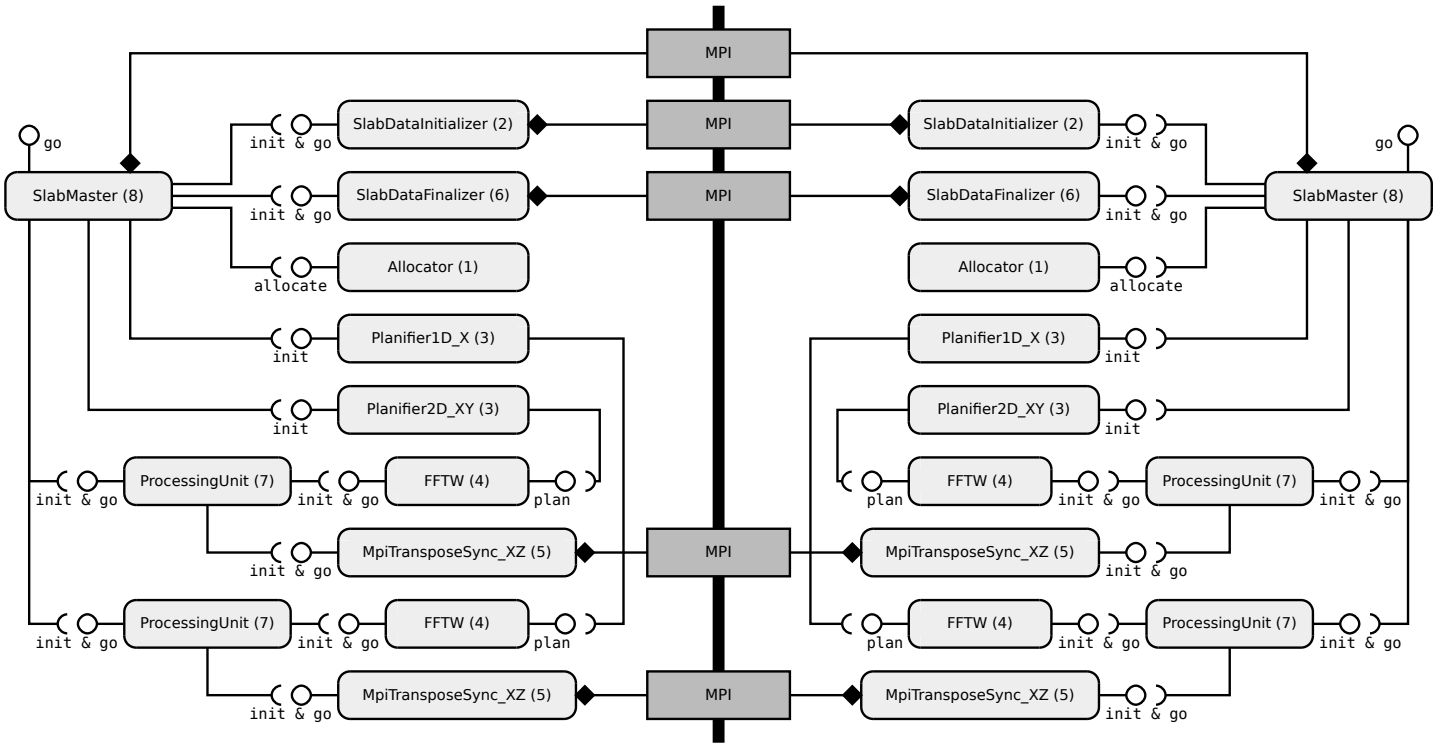



FIGURE 6.2 – Assemblage élémentaire de FFT 3D distribué sur deux processus MPI et utilisant une décomposition 1D.

## 6.3 Optimisation de l'assemblage via le remplacement de composants

Il est possible de rendre l'implémentation de FFT 3D plus efficace en remplaçant des implémentations de composants par d'autres plus optimisées, ce que l'on peut faire, par exemple avec le composant de transposition. En effet, les instances de composants peuvent être facilement

remplacées par d'autres instances qui exposent les mêmes interfaces. Par exemple, il est possible d'optimiser l'assemblage en remplaçant toutes les instances du composant `MpiTransposeSync_XZ` par des instances du composant `MpiTransposeSync_YZ`, puis en faisant de même avec les instances de `Planifier1D_X` remplacées par des instances de `Planifier1D_Y` (permettant d'appliquer des FFT 1D le long de l'axe Y, au lieu d'en appliquer selon l'axe X). Cette optimisation a été détaillée en Section 4.3 et présentée en figure 4.4. Elle rend l'accès au données plus contigu entraînant un gain sur certaines architectures matérielles durant la phase de transposition. Néanmoins, la bibliothèque de FFT doit ensuite calculer des FFTs 1D le long d'axes non contigus en mémoire impliquant éventuellement une perte de performance durant la phase de calcul. Ainsi, cette optimisation est intéressante uniquement si la perte de performance du calcul de FFT est inférieure au gain, ce qui est souvent le cas en pratique car la transposition globale est une opération très coûteuse sur les architectures distribuées disposant d'un grand nombre de cœurs. Cette optimisation a été implémentée sous le nom d'assemblage `L2C_1D1t_yz` et `L2C_1D2t_yz`, évalué au chapitre 7.

Une autre optimisation possible consiste à ne pas effectuer de transpositions locales durant les phases de transpositions globales et appliquer des FFTs le long de l'axe Z sur des données non transposées. Cela évite des copies mémoire mais ne peux être appliqué que dans le cas de la décomposition 1D avec deux transpositions. Cette optimisation est implémentée dans les assemblage `L2C_1D_2t_yz_blk` et évaluée au chapitre 7.

## 6.4 Optimisation de l'assemblage via l'adaptation des attributs

Les attributs des instances de composant peuvent être modifiées pour prendre en compte les architectures matérielles hétérogènes, tel que l'union de deux clusters différents sur Grid'5000 ou encore l'union des nœuds fin et large du supercalculateur Curie. En fait, lorsque les nœuds ne sont pas capables de calculer à la même vitesse et que les données sont équitablement distribuées entre les nœuds, le nœud le plus lent limite la vitesse du calcul si l'équilibrage de charge n'est pas correctement fait. Pour régler ce problème, un équilibrage de charge est nécessaire et les données doivent donc être non équitablement distribuées. Une solution est de contrôler la distribution des données à partir des attributs des composants. Un nouveau composant de transposition doit ainsi être implémenté afin de manipuler des données non équilibrées. Ce composant a été implémenté et les assemblages résultant (nommés `L2C_1DH_1t_yz`, `L2C_1DH_2t_yz_blk` et `L2C_2DH_3t`) sont évalués au chapitre 7.

## 6.5 Transformation globale de l'assemblage

**Réduction du nombre de transpositions** Dans de nombreux cas, la phase de transposition est la plus grande limitation, il convient donc de l'optimiser. Comme expliqué en Section 4.4, une transposition peut être évitée dans certains cas lorsqu'une décomposition 1D est utilisée (et jusqu'à deux transpositions en utilisant une décomposition 2D). Cette approche permet de supprimer la dernière transposition effectuée par l'application en adaptant l'assemblage et en ajoutant un attribut au composant maître : dans chaque processus, la seconde instance de `ProcessingUnit`, connectée au `SlabMaster` via les ports `Init` et `Go`, est ainsi supprimée, tout comme le composant de transposition qui est connecté à lui ; les port `Go` et `Init` sont directement connectés aux ports *provide* associés à l'instance de composant qui implémente la tâche 4 de la dernière phase de calcul. Étant donné que le comportement du composant `SlabMaster` diffère à l'initialisation de l'application, il est nécessaire de le réécrire. En effet, lorsqu'une transposition finale est utilisée, la manipulation des tampons mémoires, gérés par ce composant, peut changer. Un attribut de type booléen est donc ajouté au composant `SlabMaster` pour l'avertir de

la présence d'une transposition finale. Cet transformation de l'assemblage a été implémentée et évalué au chapitre 7 et est présente dans les versions `L2C_1D_1t_yz`, `L2C_1DH_1t_yz`, `L2C_2D_3t` and `L2C_2DH_3t`. La décomposition 2D actuellement implémentée permet d'éviter jusqu'à deux transpositions. Une première peut être évitée en supprimant en cascade de la dernière instance de `ProcessingUnit` (ainsi que toutes les instances connectées à ses ports *use* dû à la suppression en cascade). Une seconde peut être évitée en appliquant la même transformation que pour la décomposition 1D. L'attribut booléen est remplacé par un attribut entier afin de gérer la suppression de plus d'une transposition. Cette optimisation a été elle aussi implémentée, sous les noms `L2C_2D_3t`, `L2C_2DH_3t` et `L2C_2D_2t`.

**Recouvrement calcul/communication** Le recouvrement entre les calculs et les communications est réalisable en introduisant de nouveaux composants, en en remplaçant et en adaptant l'assemblage. En effet, le recouvrement dans notre approche et dans le cas de la décomposition 1D, peut être réalisé en répliquant les instances de `ProcessingUnit` ainsi que les instances gérant les calculs de FFT et les transpositions. Puisque les tranches de la matrice de donnée d'entrée sont traitées par plus d'instances de composant, chacune de ces instances traite moins de données que dans le cas sans recouvrement. Les attributs permettant de fixer la taille des données sont donc modifiés à cet effet. Un nouveau composant appelé `ProcessingPhase` est introduit pour acheminer les appels provenant de l'instance de `SlabMaster` aux autres instances telles que les instance de `ProcessingUnit`. Lorsqu'une méthode est appelée sur le port *provide* `Init` du composant, celui-ci appelle la même méthode avec les mêmes paramètres sur chacun des ports *provide* connectés à son port *use* `Init`. Il en est de même pour les ports `Go` de ce composant. Les interfaces du composant de transposition sont corrigées afin de permettre des opérations asynchrones et des synchronisations : le port `Go` permet ainsi de démarrer la transposition de manière asynchrone et un port `Wait` est ajouté pour permettre une synchronisation entre les instances. La synchronisation des instances est faite de manière transparente via un nouveau composant : un *adaptateur*[1] nommée `Synchronizer` forçant les instances auxquelles il est connecté via un port *use* `Wait` à être synchronisées avant ou après un appel de méthode sur le port *provide* `Go`. Un exemple d'assemblage avec recouvrement est présenté en figure 6.3. Cet assemblage a été implémenté, mais n'a pas été évalué au chapitre 7 car l'implémentation utilise des collectives asynchrones absentes dans les environnements de développement expérimentaux (c'est notamment le cas de Curie).

1. Composant dont l'instance peut être connectée à une autre (instance adaptée) et qui redirige les appels de méthodes effectués sur ses ports vers les ports de l'instance adaptée

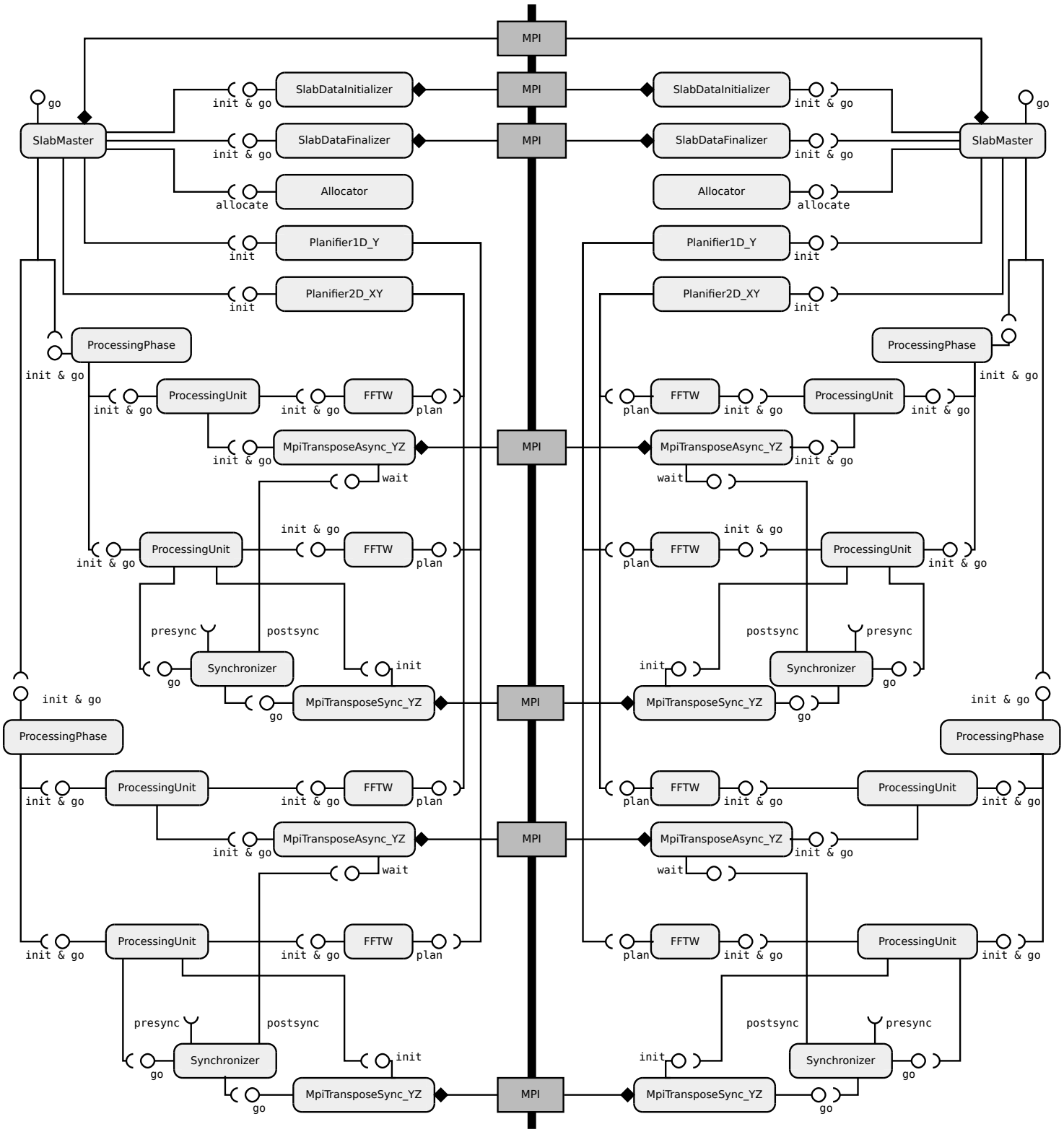



FIGURE 6.3 – Assemblage de FFT 3D distribué sur deux processus MPI recouvrant les communications par des calculs (avec deux blocs par phase) et utilisant une décomposition 1D.

**Décomposition 2D** À cause de la limite de passage à l'échelle de la décomposition 1D décrite en Section 4.3, un assemblage décrivant une décomposition 2D est nécessaire. Cela peut être fait en adaptant l'assemblage comme décrit dans la figure 6.4. Ce nouvel assemblage introduit un nouveau composant de transposition et remplace trois composants : les composants `SlabMaster`,

`SlabInitializer` et `SlabFinalizer`. Ces trois composants sont respectivement remplacés par `PencilMaster`, `PencilInitializer` et `PencilFinalizer`. Ils exposent deux nouveaux ports MPI afin que les instances d'une même ligne ou d'une même colonne dans grille de processus puissent communiquer entre elles. Ce nouvel assemblage ajoute deux trois nouvelles phases gérées par le composant `PencilMaster`, dont une de calcul et deux de transpositions (sauf si une réduction du nombre de transposition est appliquée). Puisque le schéma de décomposition 2D introduit une transposition XY des données distribuées, inutile dans le schéma de décomposition 1D, un nouveau composant de nécessite d'être développé. Cependant, le composant transposition XZ peut être réutilisé de la décomposition 1D. Cet assemblage a été implémenté et évalué au chapitre 7 sous les nom de versions `L2C_2D 3t`, `L2C_2DH_3t` et `L2C_2D_2t`.

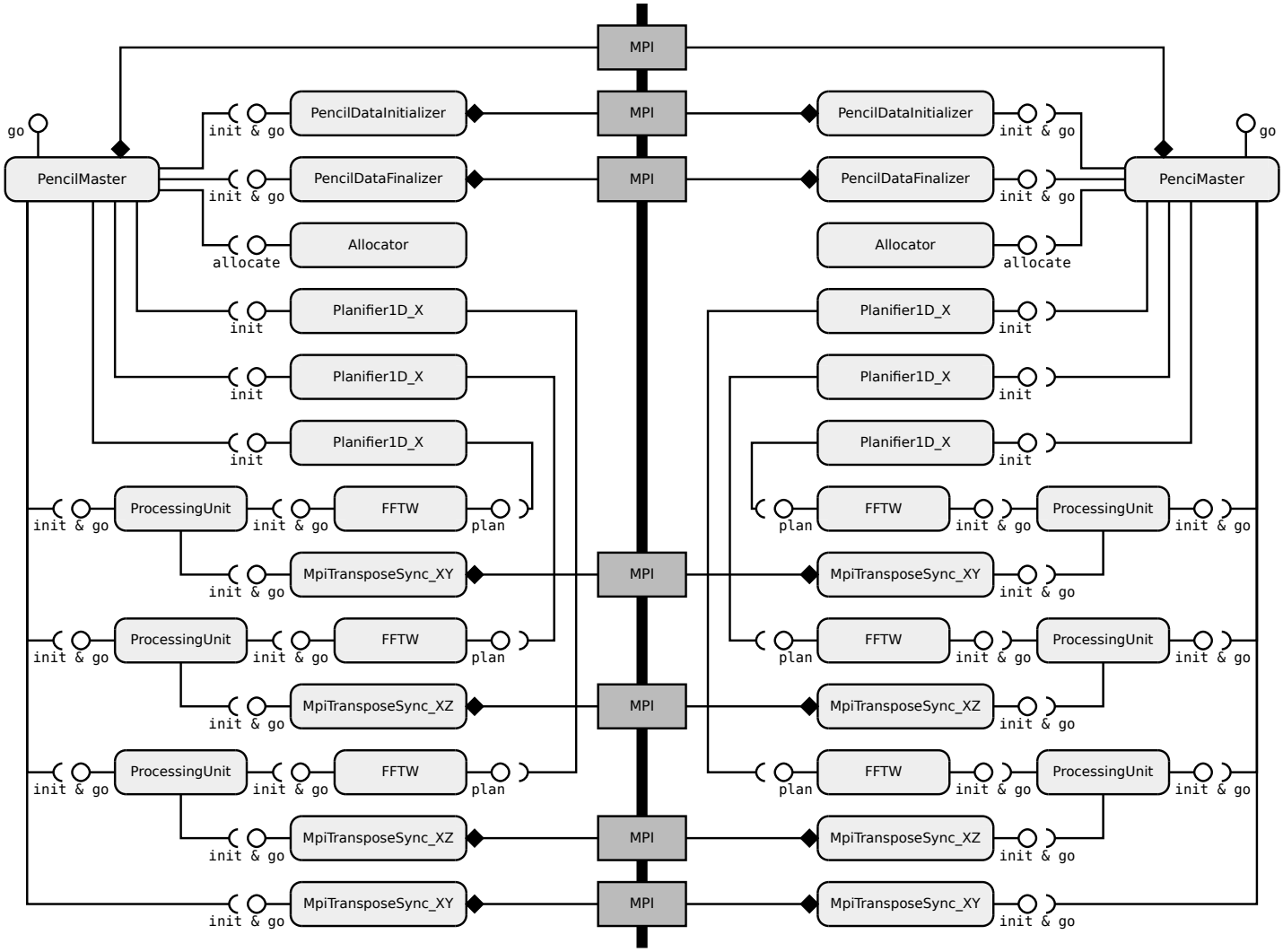



FIGURE 6.4 – Assemblage de FFT 3D distribué sur deux processus MPI (sans recouvrement) utilisant une décomposition 2D.

## 6.6 Discussion

Nous avons vu qu'il est possible de concevoir divers algorithmes de FFTs 3D en utilisant L$^2$C : à partir d'un assemblage distribué élémentaire, nous pouvons appliquer une série d'optimisations dans le but de créer un assemblage (ou variante) spécifiquement optimisé pour l'architecture matérielle sur laquelle il est exécuté dans le but de maximiser les performances sur l'architecture en

question. Tous les assemblages vus précédemment ont été implémentés. Les assemblages peuvent en pratique différer selon plusieurs critères :

— le recouvrement (*i.e.* avec ou sans) ;
— la décomposition (*i.e.* 1D ou 2D) ;
— le nombre de transpositions (*i.e.* 2, 3 ou 4) ;
— le type de transposition (*i.e.* XY, XZ ou YZ) ;
— la distribution des paramètres (*i.e.* taille des données traités par processus).

L'implémentation de tous ces assemblages a amené à la programmation de multiples composants de transposition, pouvant impacter la réutilisation de l'application. En effet, l'activation du recouvrement tout comme la variation des transpositions nécessitent tous deux d'implémenter un nouveau composant prévu à cet effet.

Pour éviter la multiplication des composants de transposition, on peut diviser le composant de transposition en trois sous-composants : un premier composant `Packer` divisant les données en plusieurs blocs contigus, ensuite redistribués entre les machines en effectuant un échange total à l'aide d'un second composant `All-to-All`, puis recomposés par un troisième composant `Unpacker`. L'instanciation du composant de transposition revient alors à instancier les trois sous-composants, puis à les interconnecter. Cette méthode permet de rendre la transposition non bloquante en réécrivant uniquement le composant `All-to-All`. De plus, il devient possible de recouvrir le traitement effectué par le composant `All-to-All` par les traitements que font les composants `Packer` et `Unpacker`. Cette transformation peut permettre d'améliorer les performances de l'application en affinant le recouvrement et d'accroître la maintenabilité de l'application en réduisant le nombre de composant à réécrire lorsqu'une optimisation est changée. Cet assemblage n'a pas été implémenté.

Afin d'évaluer les assemblages présentés, nous avons réalisé de nombreuses mesures de performance dans des situations diverses (variation de l'architecture, du nombre de cœurs, de la taille des données, etc.) et nous avons tenté d'évaluer l'adaptabilité des assemblages présentés ici.

# 7 Évaluation des performances et réutilisation

Ce chapitre vise à évaluer les performances et l'adaptabilité de certains assemblages décrits au chapitre 6. Les performances et le passage à l'échelle ont été évalués jusqu'à 8192 cœurs et sur des architectures autant homogènes qu'hétérogènes. L'adaptabilité se réfère à la capacité d'une implémentation à intégrer de nouvelles optimisations diverses et à réutiliser le code d'autres implémentations.

## 7.1 Évaluation des performances

### 7.1.1 Configuration de l'environnement et méthodologie

**Architectures matérielles utilisées** Un premier groupe d'essais a été effectué sur la plateforme expérimentale Grid'5000 (figures 7.2, 7.3, 7.4 et 7.5, 7.7 et 7.8) dont un aperçu est donné au chapitre 1. Pratiquement tous les clusters ont été utilisés comme supports d'expérimentation. La figure 8.1 liste toutes les grappes de serveurs utilisées et donne quelques informations complémentaires sur le matériel dont ils sont constitués. Les expériences hétérogènes utilisent les grappes Genepi et Edel. Ces deux grappes sont connectées à un même réseau InfiniBand, mais elles ont des processeurs différents ce qui les rend intéressantes pour effectuer des expériences hétérogènes. Un second groupe d'essais, dont le but était d'évaluer le passage à l'échelle des assemblages, a été réalisé sur le supercalculateur Curie (figures 7.9, et 7.10).

**Algorithmes et implémentations de référence** La figure 7.1 résume les assemblages $L^2C$ et bibliothèques de références utilisées dans les expériences. Toutes les implémentations des expériences appliquent des transformations de nombres complexes vers des nombres complexes. Les deux types de décompositions (1D et 2D) ont été évaluées lorsque c'est possible. Les implémentations sur Grid'5000 et proposant une décomposition 1D ont été évaluées avec et sans transposition finale. Les implémentations utilisant le recouvrement n'ont pas été évaluées dû à la difficulté de mettre en œuvre les expériences (comme décrit en section 6.5).

| Nom de l'assemblage | Décomposition | Nombre de transpositions | Support de l'hétérogénéité |
|---|---|---|---|
| L2C_1D_2t_xz | 1D | 2 | non |
| L2C_1D_1t_yz | 1D | 1 | non |
| L2C_1D_2t_yz | 1D | 2 | non |
| L2C_1D_2t_yz_blk | 1D | 2 | non |
| L2C_1DH_1t_yz | 1D | 1 | oui |
| L2C_1DH_2t_yz_blk | 1D | 2 | oui |
| L2C_2D_4t_xyz | 2D | 4 | non |
| L2C_2D_3t_xzy | 2D | 3 | non |
| L2C_2D_3t_yxz | 2D | 3 | non |
| L2C_2D_2t_yzx | 2D | 2 | non |
| L2C_2D_2t_zxy | 2D | 2 | non |
| L2C_2D_3t_zyx | 2D | 3 | non |
| L2C_2DH_2t_zxy | 2D | 2 | oui |
| **Nom de la bibliothèque** | **Décomposition** | **Nombre de transpositions** | **Support de l'hétérogénéité** |
| FFTW | 1D | 2 | non utilisé |
| FFTW_1t | 1D | 1 | non utilisé |
| 2DECOMP_1D1t | 1D | 1 | non disponible |
| 2DECOMP_1D2t | 1D | 2 | non disponible |
| 2DECOMP_2D | 2D | 2 | non disponible |

FIGURE 7.1 – Assemblages et bibliothèques utilisées dans les expériences.

Les bibliothèques de FFT utilisées comme références sont la FFTW 3.3.4 et 2DECOMP 1.5. P3DFFT n'a pas été utilisée car la transformation de nombres complexes vers des nombres complexes n'a pas été encore implémentée or cette étude se concentre uniquement sur cette transformation. SPIRAL n'a pas été utilisé comme bibliothèque de référence car il n'a été découvert que tardivement au cours du stage et la version MPI qui nous intéresse ne semble pas être disponible publiquement actuellement. Toutes les bibliothèques et assemblages ont été configurés pour utiliser des transformations de nombres complexes vers des nombres complexes (en double précision), sans recouvrement et se basent sur la FFTW comme bibliothèque séquentielle (configurée pour produire des plans de type FFTW_MEASURE). Le compilateur utilisé pour compiler les implémentations sur Grid'5000 est gcc (en version 4.7.2) et l'implémentation MPI utilisée est OpenMPI (en version 1.8.1). Sur Curie, le compilateur Intel C++ (en version 1.2.7.2) a été utilisé pour la compilation. L'implémentation MPI utilisée est Bullmpi, basé sur OpenMPI.

**Méthodologie d'évaluation** Un premier groupe d'expériences (figures 7.2, 7.3, 7.9, et 7.10) a été faite dans un cadre homogène afin d'analyser les performances et le passage à l'échelle des assemblages. Un second groupe d'expériences (figures 7.4, 7.5, 7.7 et 7.8) a été réalisé pour analyser les performances des assemblages dans un cadre hétérogène, mais aussi afin d'étudier l'impact de la variation de certains paramètres de l'assemblage sur les performances. Les expériences ont été fait 100 fois et la moyenne des temps a été sélectionnée comme descripteur des performances. La moyenne n'étant pas suffisante pour décrire les performances lorsque la variance des résultats est grande, des barres d'erreurs correspondant respectivement au premier et dernier quartile ont été placées sur les figures. On notera toutefois que les barres d'erreur sont très petites sur la plupart des courbes présentées. Toutes les matrices sont cubiques ($256^3$, $512^3$, $1024^3$, etc.). La taille des matrices d'entrées ainsi que le nombre de processus sont des puissances de deux.

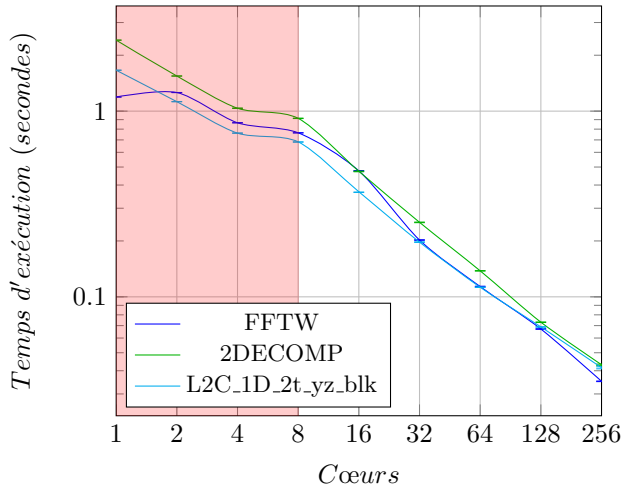
Dans le cas hétérogène et en utilisant une décomposition 1D, les données sont distribuées selon l'axe Z et la taille des tranches dépend des performances des nœuds : sur chaque nœud, le temps nécessaire pour appliquer une seule FFT 3D (en utilisant tous les cœurs) est récupéré expérimentalement et la taille des blocs est fixée de manière inversement proportionnelle au

temps de calcul. En utilisant une décomposition 2D, la même opération est appliquée le long de l'axe Z et le découpage le long de l'axe Y reste inchangé par rapport au cas homogène.

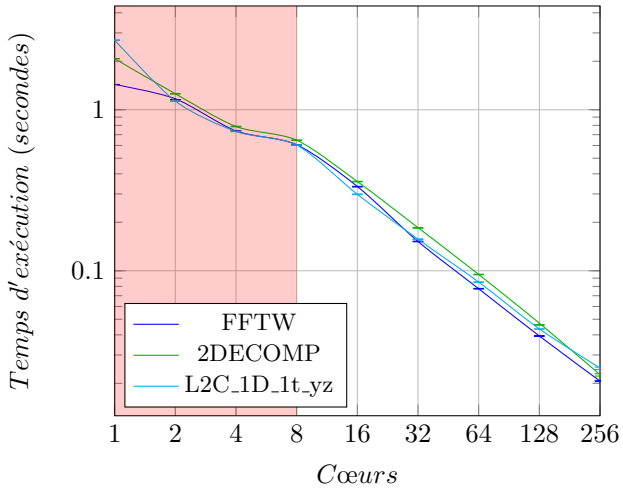
Afin d'évaluer l'impact de certains paramètres tels que la taille des données, le nombre de transpositions, etc. sur les performances des assemblages, nous avons utilisé la bibliothèque Execo [45] permettant de réaliser un large ensemble d'expériences facilement sur la plate-forme Grid'5000.

### 7.1.2 Expériences homogènes

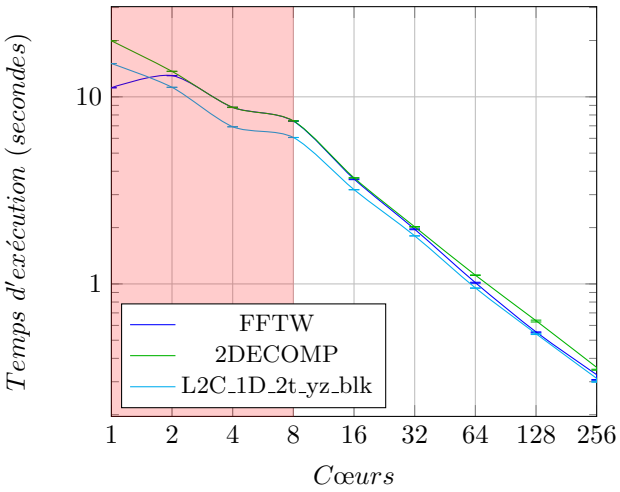
La figure 7.2 présente les résultats obtenus sur la grappe Griffon jusqu'à 256 cœurs, pour une décomposition 1D, avec des matrices de taille $256^3$ (figure 7.2a et figure 7.2b) et $512^3$ (figure 7.2c et figure 7.2d) avec et sans transposition supplémentaire.



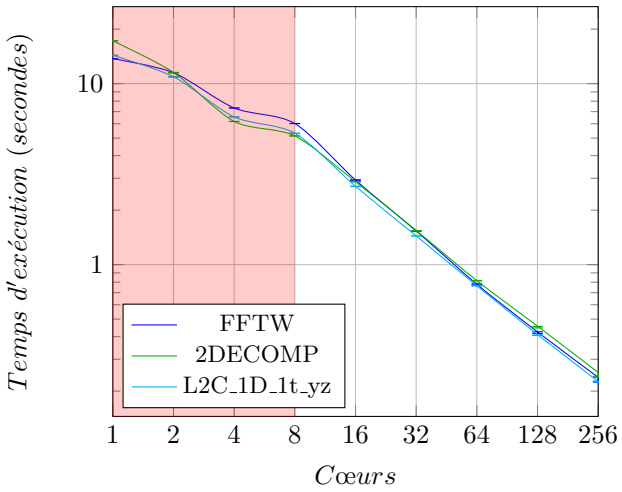

(a) avec une matrice de taille $256^3$ avec 2 transpositions.




(b) avec une matrice de taille $256^3$ avec 1 transpositions.




(c) avec une matrice de taille $512^3$ avec 2 transpositions.




(d) avec une matrice de taille $512^3$ avec 1 transpositions.

FIGURE 7.2 – Temps d'exécution d'une FFT 3D (de type complexes vers complexes) homogène sur la grappe Griffon faisant intervenir une décomposition 1D. La partie rouge désigne les résultats obtenus sur un seul nœud, alors que la partie blanche les résultats obtenues sur plusieurs nœuds.

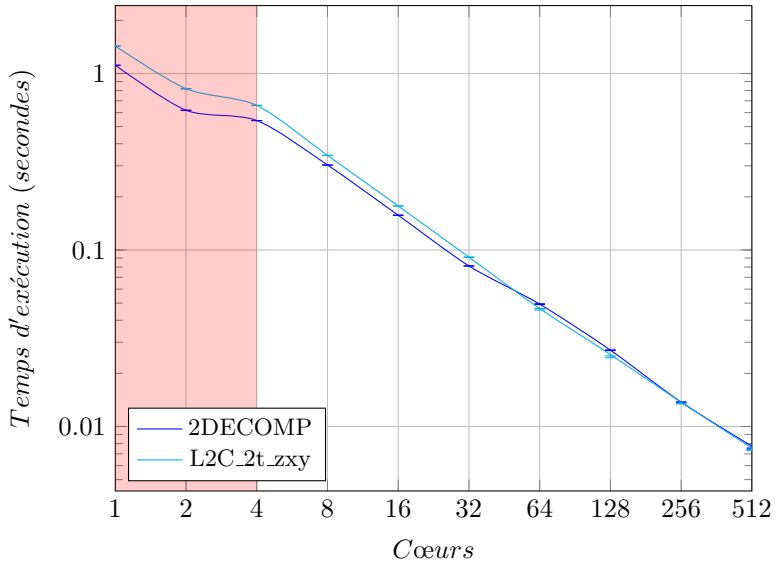



FIGURE 7.3 – Temps d'exécution d'une FFT 3D (de type complexes vers complexes) homogène avec une matrice de taille $256^3$ sur la grappe Graphene faisant intervenir une décomposition 2D et n'utilisant que deux transpositions.

Dans l'ensemble, les résultats des performances de $L^2C$, de la FFTW et de 2DECOMP sont similaires. On peut noter quelques exceptions sur Grid'5000 :

- sur 1 cœur (séquentiel), la FFTW est meilleur ;
- 2DECOMP est un peu moins performant que $L^2C$ et FFTW sur des matrices de taille $256^3$ ;
- la FFTW est 20% plus rapide que $L^2C$ et 2DECOMP sur 256 cœurs avec des matrices de taille $256^3$ et en utilisant deux transpositions.

La variation des performances que l'on peut observer avec la FFTW provient de la sélection d'un algorithme rapide durant la phase de planification. Lorsque le planificateur utilise le mode FFTW_MEASURE, la FFTW utilise une heuristique pour trouver un algorithme performant mais ne trouve pas nécessairement le meilleur. Cela peut être résolu en utilisant plutôt le mode FFTW_EXHAUSTIVE qui réalise une exploration exhaustive de tous les algorithmes qu'il peut produire. En contrepartie, ce mode rend la planification souvent très lente.

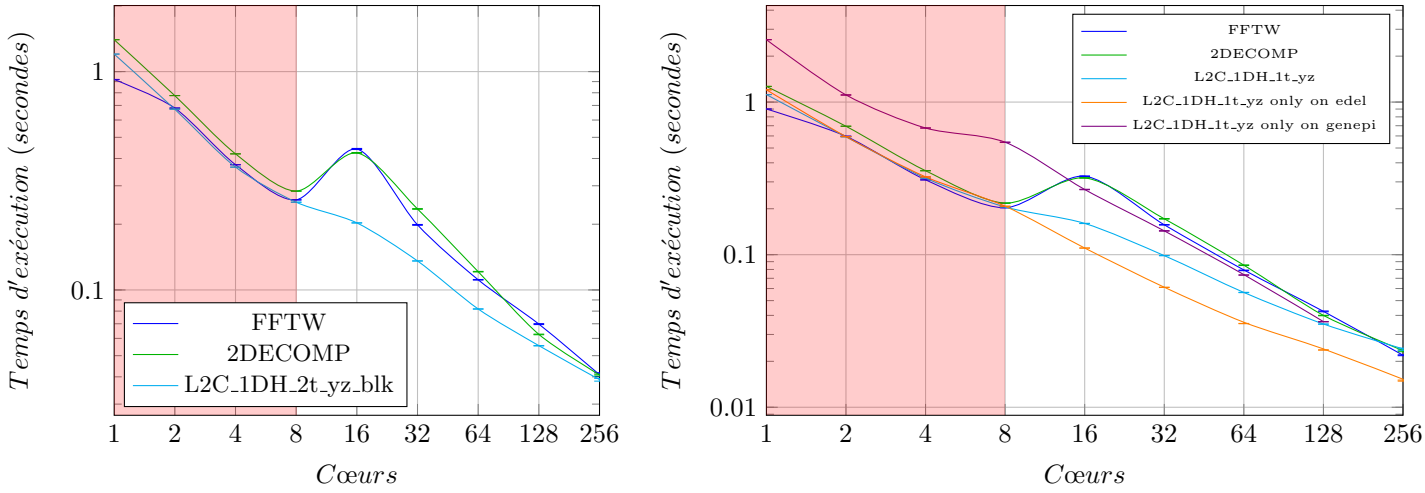
La figure 7.3 présente les résultats sur la grappe Graphene jusqu'à 512 cœurs et faisant intervenir une décomposition 2D (`L2C_2D_2t` et 2DECOMP_2D) sur des matrices de tailles $256^3$. Les résultats montrent que l'assemblage $L^2C$ est 28% plus lent que 2DECOMP sur un seul cœur, mais au-delà d'un nœud utilisé, l'assemblage $L^2C$ est compétitif avec 2DECOMP car l'écart de performance entre les deux est très faible voir quasi nul lorsque le nombre de cœurs s'approche 512.

### 7.1.3 Expériences de variation de paramètres

#### Expérimentations hétérogènes

La figure 7.4 présente les résultats sur les grappes Edel et Genepi jusqu'à 256 cœurs en utilisant une décomposition 1D, avec une matrice de taille $256^3$, avec et sans transposition supplémentaire. Les résultats jusqu'à 8 cœurs correspondent au cas hétérogène puisque les cœurs appartiennent tous au nœud Edel. À partir de 16 cœurs, la moitié des cœurs proviennent d'Edel et l'autre de Genepi. La grappe Edel est globalement plus rapide que Genepi pour effectuer des calculs de FFT 3D.

On peut voir que de 8 à 16 cœurs, les performances de 2DECOMP diminuent. Cela vient du fait que 2DECOMP n'effectue aucun équilibrage de charge et est donc limité par les nœuds les plus lents (ici les nœuds de Genepi).




(a) avec une matrice de taille $256^3$ avec 2 transpositions.

(b) avec une matrice de taille $256^3$ avec 1 transposition.

FIGURE 7.4 – Temps d'exécution d'une FFT 3D (de type complexes vers complexes) hétérogène sur Edel et Genepi faisant intervenir une décomposition 1D

La figure 7.4b montre que la performance de l'assemblage $L^2C$ se trouve entre les performances obtenues uniquement sur Edel (la grappe la plus rapide ici) et les performances obtenues uniquement sur Genepi (la grappe la plus lente ici). Cela signifie que l'assemblage $L^2C$ tire bien avantage des performances offertes par les deux grappes et n'est pas uniquement limité par le plus lent. Néanmoins, lorsque le nombre de cœurs tend vers 256 (la limite permise par la décomposition 1D dans cette expérience) les performances de l'assemblage hétérogène se rapprochent de celles de 2DECOMP et de celles de l'assemblage exécuté uniquement sur les nœuds de Genepi. Cela provient de la décomposition 1D utilisée. En effet, la hauteur des tranches tend vers 1 et l'arrondissement de la taille des tranches à des valeurs entières empêche d'effectuer un équilibrage de charge efficace. Les performances obtenues uniquement sur les nœuds Genepi et uniquement sur les nœuds Edel ont été reportées sur les graphes et ont un comportement similaire en terme de passage à l'échelle. C'est en grande partie dû à l'échange total qui est aussi rapide sur les deux grappes car elles utilisent toutes les deux le même réseau d'interconnexion (*i.e.* InfiniBand 40G).

Globalement, l'assemblage $L^2C$ hétérogène est au moins aussi rapide (jusqu'à 117% plus rapide sur 16 cœurs avec une transposition supplémentaire) que 2DECOMP car il supporte des distributions hétérogènes de données.

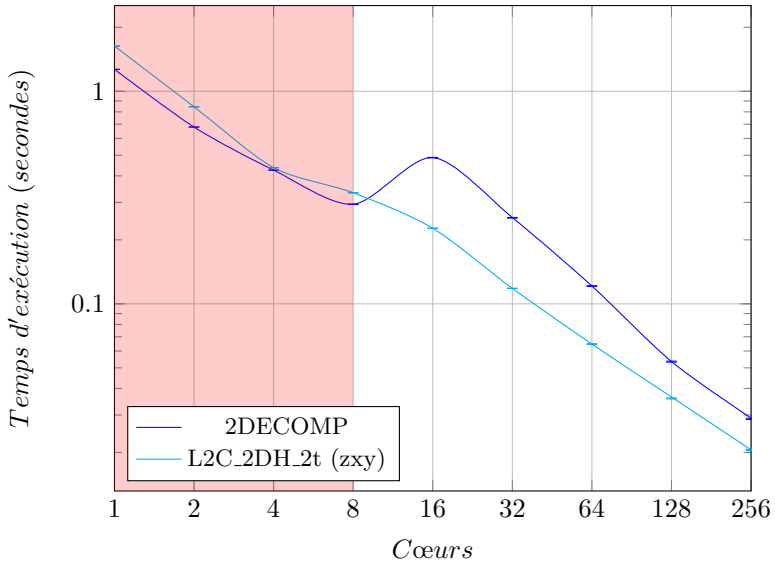



FIGURE 7.5 – Temps d'exécution d'une FFT 3D (de type complexes vers complexes) hétérogène sur une matrice de taille $256^3$ sur Edel et Genepi faisant intervenir une décomposition 2D et minimisant le nombre de transposition.

La figure 7.5 montre les résultats sur Edel et Genepi de 1 à 256 cœurs, utilisant une décomposition 2D, et des matrices de taille $256^3$, sans transpositions supplémentaires. Les performances de l'assemblage hétérogène `L2C_2DH_2t` ont un comportement similaire à ce qu'on peut observer lors de l'usage d'une décomposition 1D. Les résultats montre que si plusieurs nœuds sont utilisés, l'assemblage $L^2C$ est toujours plus rapide que 2DECOMP. En effet, la décomposition 2D permet d'obtenir une répartition de charge de manière plus fine que la décomposition 1D (en supprimant quasiment le problème des arrondis de taille de blocs) et cela à un impact sur les performances de l'assemblage $L^2C$. Cela montre donc l'intérêt d'un équilibrage de charge sur les architectures hétérogènes. Ainsi, l'assemblage est 115% plus rapide que 2DECOMP sur 16 cœurs et 41% plus rapide sur 256 cœurs.

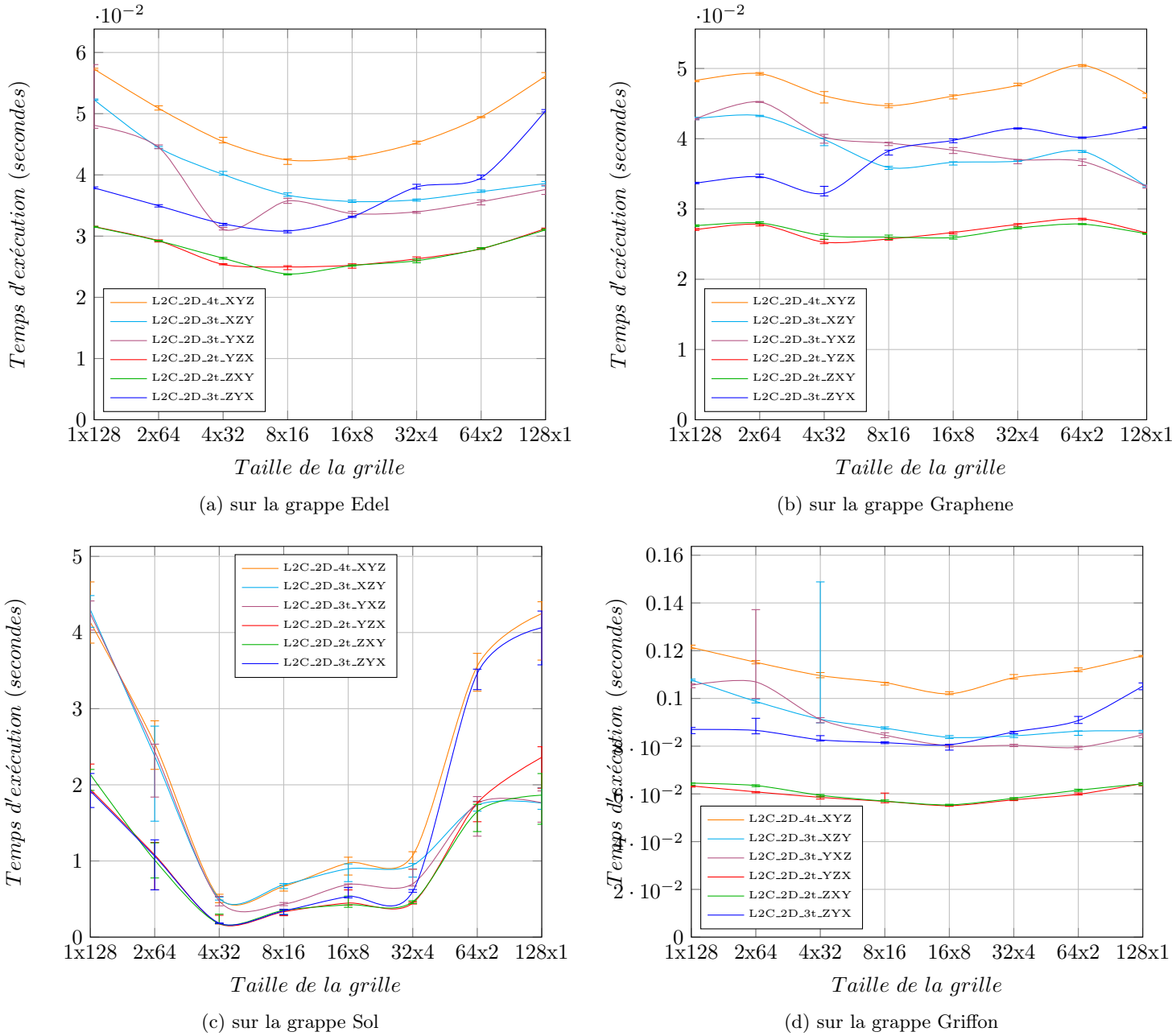
### Distributions des données, transpositions et architectures

Tout comme l'assemblage peut être spécialisé pour répondre à un problème d'équilibrage de charge, il existe d'autres cas dans lesquels l'assemblage peut être adapté pour offrir de meilleures performances. C'est en particulier le cas lorsqu'une décomposition 2D est utilisée : le découpage des données entre les processus (grille de processus) peut influer sur les performances et dépendre lui même d'autres paramètres tels que l'architecture matérielle, la taille des données, les transpositions utilisées, etc. Nous avons donc dans un premier temps analysé les performances obtenues sur Edel en fonction de la taille de la grille de processus choisie et des transpositions réalisées. Puis, nous avons étendu notre étude à plus architectures matérielles en réalisant des expériences sur les grappes Griffon, Sol et Graphene. Enfin, nous avons finalement réalisé de nouvelles expériences dans lesquelles le nombre de cœurs a été ajusté.

Les assemblages évalués sont associés à une certaine organisation de la matrice de sortie. Étant donnée qu'il y a 6 manières d'organiser la matrice de sortie avec une décomposition 2D, six assemblages ont été évalués. La figure 7.6 montre pour chaque assemblage les transpositions qu'il applique et l'organisation de la matrice de sortie. La matrice d'entrée est supposée être organisée respectivement selon les axes X, Y et Z.

| Nom d'assemblage | Transpositions appliquées | Organisation des données en sortie |
|---|---|---|
| `L2C_2D_4t_xyz` | XZ + XY + XY + XZ | XYZ |
| `L2C_2D_3t_xzy` | XZ + XY + XZ | XZY |
| `L2C_2D_3t_yxz` | XY + XZ + XZ | YXZ |
| `L2C_2D_2t_yzx` | XZ + XY | YZX |
| `L2C_2D_2t_zxy` | XY + XZ | ZXY |
| `L2C_2D_3t_zyx` | XZ + XY + XY | ZYX |

FIGURE 7.6 – Description de l'ensemble des assemblages utilisés dans les expérimentations.




(a) sur la grappe Edel

(b) sur la grappe Graphene

(c) sur la grappe Sol

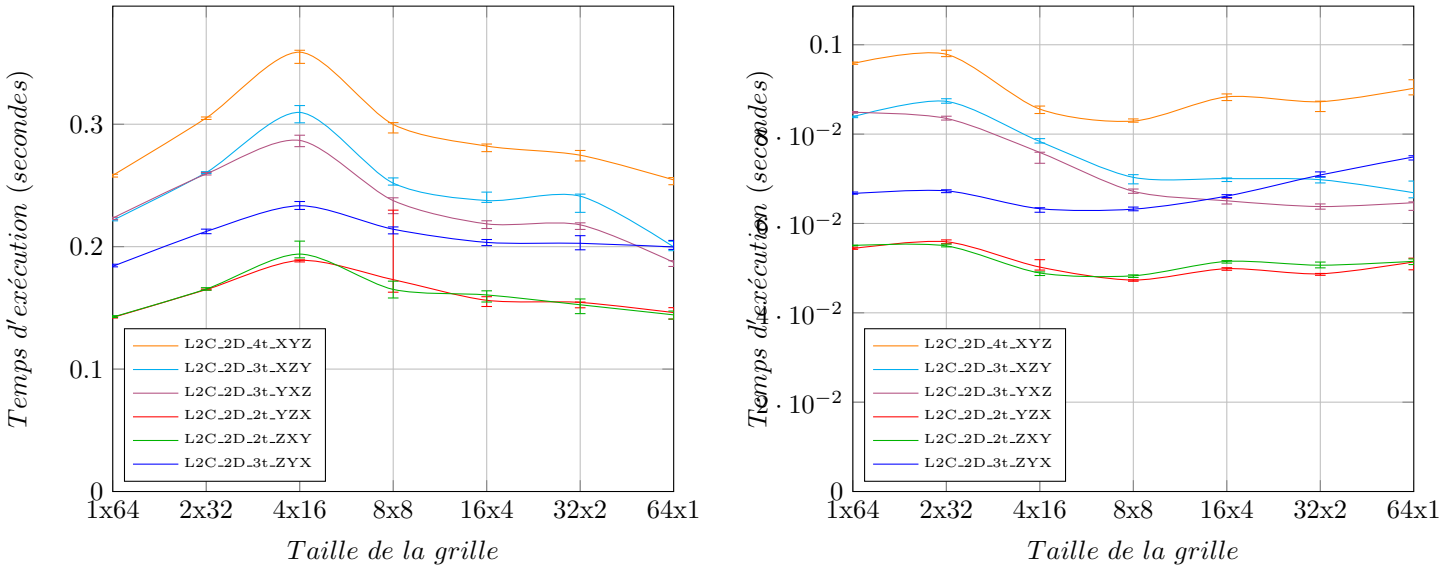
(d) sur la grappe Griffon

FIGURE 7.7 – Temps d'exécution d'une FFT 3D (de type complexes vers complexes) homogène avec des matrices de taille $256^3$, en utilisant cœurs et faisant intervenir une décomposition 2D.

La figure 7.7a montre les performances d'assemblages L$^2$C appliquant chacun une liste de transposition différente, sur la grappe Edel, en utilisant 128 cœurs, en fonction de la taille de la

grille de processus choisie. Les résultats montrent que les performances optimales sont souvent obtenues lorsque la taille de la grille est proche de $\sqrt{p} \times \sqrt{p}$, où $p$ est le nombre de cœurs utilisé dans l'expérience (en l'occurrence 128). Néanmoins, dans le cas de l'assemblage `L2C_2D_3t_yxz`, les performances optimales sont obtenues en utilisant une grille de taille 4. On peut aussi voir que les assemblages `L2C_2D_2t_yzx` et `L2C_2D_2t_zxy` ont globalement les mêmes performances. En effet, ces deux assemblages ne diffèrent que de l'ordre des transpositions qu'ils appliquent. On peut donc conclure que la taille de la grille de processus qui maximise les performances est impactée par les transpositions effectués.

Les figures 7.7b, 7.7c et 7.7d présentent les résultats de la même expérience, mais cette fois-ci en utilisant respectivement les grappes Graphene, Sol et Griffon. On peut observer des variations de performances d'une grappe à une autre et des comportement différents. En effet, les résultats montrent que la taille de grille $128 \times 1$ maximise les performances des assemblages `L2C_2D_3t_xzy` et `L2C_2D_3t_yxz` et qui est bien loin de $\sqrt{p} \times \sqrt{p}$. Sur sol, les performances des assemblages s'effondrent lorsqu'un trop petit nombre de cœurs est utilisé le long d'un des deux axes de la grille. Cela provient d'un phénomène de congestion réseau : chaque processus envoi un trop grand nombre de messages saturant ainsi les commutateurs présent sur le réseau et allongeant le temps de complétion de l'échange total. Utiliser une grille de processus de taille $4 \times 32$ semble être un bon choix quelque soit la transposition utilisée sur la grappe Sol. De même, sur la grappe Griffon, une grille de taille $16 \times 8$ semble être un bon choix. On peut donc conclure que l'architecture matérielle impacte grandement la taille de la grille de processus qui maximise les performances.




(a) sur la grappe Griffon

(b) sur la grappe Graphene

FIGURE 7.8 – Temps d'exécution d'une FFT 3D (de type complexes vers complexes) homogène avec des matrices de taille $256^3$, en utilisant 64 cœurs et faisant intervenir une décomposition 2D.

Sur certaines grappes telles que Griffon, le nombre de processus impacte les tendances des performances observées jusqu'ici. La figure 7.8a présente les résultats de la même expérience, mais en utilisant cette fois si que 64 cœurs sur les grappes Griffon et Graphene. Les résultats sur la grappe Griffon montre que l'utilisation d'une taille de grille proche de $\sqrt{p} \times \sqrt{p}$ ne donne pas de bonnes performances quelque soit les transpositions utilisées. Au contraire, les tailles de la grille qui maximise les performances sont $1 \times 64$ et $64 \times 1$ qui reviennent tous les deux à appliquer une décomposition 1D (et ainsi à réduire le nombre de transpositions distribuées). Néanmoins, ce phénomène n'est pas présent sur toutes les architectures. En effet, sur la grappe Graphene, les

résultats sont similaires à ceux obtenus sur 128 cœurs. On peut déduire que le nombre de cœurs utilisés influe lui aussi sur le comportement des performances des assemblages L$^2$C.

L'assemblage à utiliser parmi les 6 présentés précédemment peut être sélectionné en fonction des contraintes fixées par l'utilisateur de l'application : si aucune contrainte sur l'organisation de la matrice de sortie n'est fixée, l'assemblage minimisant le nombre de transposition est choisi (les assemblages `L2C_2D_2t_yzx` ou `L2C_2D_2t_zxy`, au choix, puisqu'il sont aussi performants), autrement l'assemblage proposant l'organisation de la matrice de sortie demandée est sélectionné.

Nous avons vu que la taille de la grille de processus MPI influe sur les performances des assemblages tout comme le choix de l'organisation de la matrice de sortie. Afin de maximiser les performances des assemblages, il est nécessaire de choisir la taille de la grille en fonction de ces critères. Étant donnée la variabilité des architectures parallèles et leur évolution rapide, il est préférable de choisir automatiquement la taille de la grille automatiquement. Cela peut se faire via un algorithme de choix qui à partir d'une modélisation de l'architecture matérielle et des paramètres applicatif sélectionnerait la bonne taille de grille.

L'algorithme de choix peut alors être intégré dans un modèle de composants de haut niveau tel que HLCM : des composants génériques seraient spécialisés en fonction des résultats produit par l'algorithme de choix générant ainsi un assemblage de plus bas niveau comme ceux que nous avons présentés au chapitre 6.

### 7.1.4 Passage à l'échelle

La figure 7.9 présente les résultats obtenus sur les nœuds fins [1] du supercalculateur Curie de 32 à 1024 cœurs en utilisant une décomposition 1D, pour des matrices de taille $1024^3$ et sans transpositions supplémentaires. Les performances de L$^2$C, FFTW et DECOMP sont similaires aux performances obtenues sur Grid'5000 et on peut voir que l'assemblage L$^2$C est légèrement plus lent que 2DECOMP.

La figure 7.10a montre les résultats obtenus sur les nœuds fins du supercalculateur Curie de 256 à 8192 cœurs en utilisant une décomposition 2D, pour des matrices de taille $1024^3$ et sans transpositions supplémentaires. Les performances de l'assemblage sont similaires celles obtenues sur Grid'5000 en terme de passage à l'échelle. Donc l'assemblage passe à l'échelle jusqu'à 8192 cœurs. Cependant, il est globalement plus lent que 2DECOMP quel que soit le nombre de cœurs utilisés ici, bien que les performances entre les deux tendent à être identique lorsque le nombre de cœurs s'approche de 8192. L'écart s'échelonne de 41% sur 256 cœurs à 21% sur 8192 cœurs.

La figure 7.10b présente les résultats de la même expérience mais en utilisant de plus grandes tailles de matrice ($4096^3$) et entre 2048 et 8192 cœurs. Les résultats sont proches de l'expérience précédente et confirment aussi que l'assemblage passe bien à l'échelle avec de plus grandes matrices. Néanmoins, l'assemblage L$^2$C tend à être moins performant que 2DECOMP lorsque le nombre de cœurs s'approche de 8192. En effet, l'assemblage est 18% plus lent que 2DECOMP sur 2048 cœurs, 30% sur 4096 cœurs et 33% sur 8192 cœurs. Il faudrait mener des expériences avec plus de cœurs pour confirmer cette hypothèse.

Au-delà de la limite des 8192 cœurs, l'implémentation actuelle de L$^2$C prend trop de temps à déployer l'assemblage sur les nœuds et la quantité de mémoire utilisée pour effectuer le déploiement devient trop grande. Cela provient principalement de l'augmentation de la taille du fichier d'assemblage lorsque le nombre de cœurs grandit et de la lecture intensive des fichiers contenant la description binaire des composants (*i.e.* bibliothèques dynamiques). En effet, le fichier de description d'assemblage lu par L$^2$C a une taille proportionnelle à la taille de l'assemblage. Or la lecture de l'assemblage L$^2$C est faite séquentiellement dans chaque processus MPI et elle garde

1. Possédant deux processeur Intel 8 cœurs donc 16 cœurs par nœud.

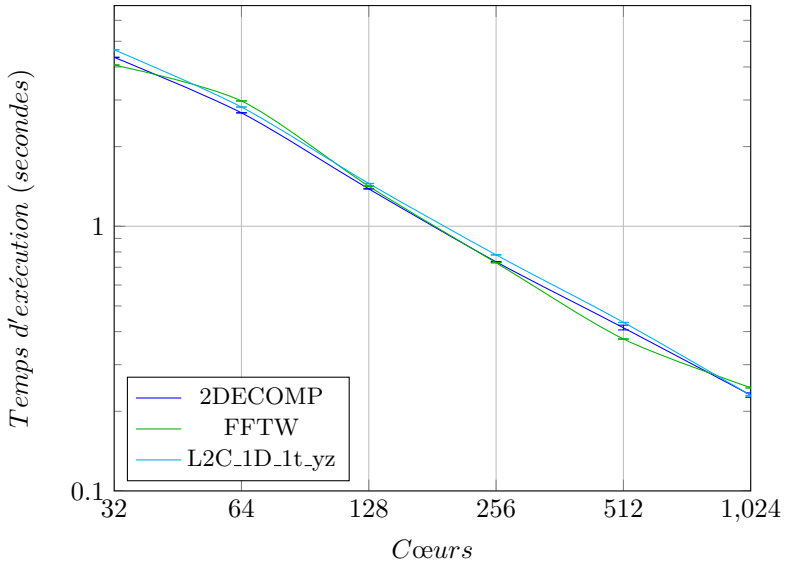



FIGURE 7.9 – Temps d'exécution d'une FFT 3D (de type complexes vers complexes) homogène sur une matrice de taille $1024^3$ sur Curie faisant intervenir une décomposition 2D et minimisant le nombre de transpositions

une trace mémoire du XML ce qui pose irrémédiablement problème lorsqu'un trop grand nombre de cœurs sont utilisés. De plus, un autre frein au passage à l'échelle apparaît : les bibliothèques présentes sur un unique système de fichiers sont lues par un grand nombre de processus et le surcharge rapidement.

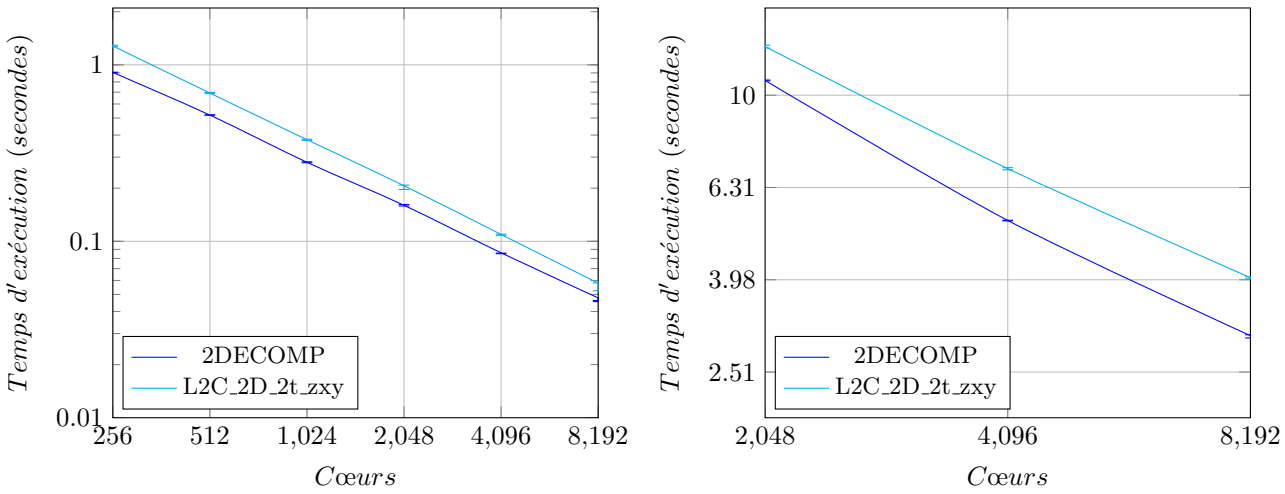
Afin de régler ces problèmes, nous planifions de lire le XML en gardant une trace mémoire minimisée sur l'ensemble de la structure du fichier afin de garder une empreinte mémoire faible lorsqu'un grand nombre de cœurs sont utilisés. Nous prévoyons aussi de réunir l'ensemble des bibliothèques dynamiques nécessaires au bon fonctionnement de l'application au sein d'un unique package qui pourrait être distribué automatiquement par MPI de manière efficace.

## 7.2 Évaluation de la réutilisation

Cette section évalue l'adaptabilité de l'approche basée sur des composants $L^2C$ comparée aux bibliothèques de références précédemment sélectionnées.

### Comparaison des optimisations

Le tableau 7.11 montre une comparaison d'optimisations implémentées ou possibles entre l'assemblage et les bibliothèques de références. Les assemblages peuvent être adaptés pour effectuer toutes les optimisations vues précédemment. Pour appliquer des optimisations spécifiques aux supercalculateurs Cray XTs, l'assemblage peut être adapté afin d'utiliser un composant de transposition spécifique qui rembourre les tampons avec des zéros et utiliser un `MPI_Alltoallv` ou peut sinon réaliser un échange total en utilisant la mémoire partagée proposée par cette famille de supercalculateurs. Afin d'effectuer un échange total entre les nœuds, la FFTW dispose de plusieurs algorithmes de transposition et en sélectionne le plus rapide durant la planification.




(a) avec une matrice de taille $1024^3$

(b) avec une matrice de taille $4096^3$

FIGURE 7.10 – Temps d'exécution d'une FFT 3D (de type complexes vers complexes) homogène sur Curie faisant intervenir une décomposition 2D et minimisant le nombre de transposition

| | FFTW | P3DFFT | 2DECOMP | Assemblage |
|---|---|---|---|---|
| Décomposition 1D | Oui | Oui | Oui | Oui |
| Décomposition 2D | Non | Oui | Oui | Oui |
| Équilibrage de charge (nœuds hetero.) | Oui *(manuellement)* | Non | Non | Oui *(en adaptant l'assemblage)* |
| Recouvrement des communications | Non | Oui *(limité à des direction et tailles de blocs fixées)* | Oui | Oui *(en adaptant l'assemblage)* |
| Padding pour éviter un `MPI_Alltoallv` | Non | Oui | Oui | Possible *(avec de nouveaux composants de transposition)* |
| Utilisation de la mémoire partagée des supercalculateurs Cray XTs | Non | Oui | Oui | Possible *(avec de nouveaux composants de transposition)* |
| Méthode d'échange total | Quelques algorithmes de transposition MPI | Basé sur un échange MPI all-to-all | Basé sur un échange MPI all-to-all | Plusieurs versions possibles *(version du all-to-all MPI implémentée)* |

FIGURE 7.11 – Comparaison d'optimisations implémentées/possibles entre l'assemblage et les bibliothèques de références

| Version | Lignes de code C++ | Code réutilisé |
|---|---|---|
| `L2C_1D_2t_xz` | 927 | - |
| `L2C_1D_1t_yz` | 929 | 77% |
| `L2C_1D_2t_yz` | 929 | 100% |
| `L2C_1D_2t_yz_blk` | 1035 | 69% |
| `L2C_1DH_1t_yz` | 983 | 80% |
| `L2C_1DH_2t_yz_blk` | 1097 | 72% |
| `L2C_2D_4t_xyz` | 1067 | 87% |
| `L2C_2D_2t_zxy` | 1067 | 100% |
| `L2C_2DH_2t_zxy` | 1146 | 69% |

FIGURE 7.12 – Nombre total de lignes de code pour les variantes de l'application de FFT 3D et pourcentage de code réutilisé des assemblages qui précèdent dans le tableau.

### Réutilisation

Le tableau 7.12 montre la réutilisation du code (en terme de nombre de lignes de code C++) entre quelques assemblages L$^2$C. La réutilisation est la quantité de code qui est réutilisée des assemblages énoncés plus haut dans le tableau. Globalement, notre implémentation L$^2$C est plus courte que celle de 2DECOMP ou P3DFFT (contenant respectivement 11570 et 8118 lignes de code FORTRAN) ; c'est parce que 2DECOMP et P3DFFT implémentent plus de fonctionnalités.

Étant donné que les composants sont de grain moyen et qu'il exposent des interfaces simples (voir chapitre 6), modifier un assemblage pour un unique PE revient uniquement à modifier quelques paramètres, connexions et à ajouter/supprimer des instances. Ce processus n'implique aucune modification de bas niveau (*e.g.* changement de code interne aux composants, programmation de nouveaux composants, etc.). Tout est relayé à la manipulation de l'assemblage et est indépendant des possibles changements internes aux implémentations de composants.

Avec L$^2$C, la description d'assemblage a besoin d'être réécrite pour chaque architectures matérielle ciblée ou à chaque modification des paramètres applicatifs (tel que la taille des données traitées). Comme ce processus est délicat et est sujet à des erreurs, de telles descriptions devraient être automatiquement générées. Bien que cela soit fait actuellement par un groupe de scripts, l'adaptation de ces programmes peut être fastidieux et leur maintenance reste problématique. Un modèle de composants de plus haut niveau automatisant la génération de l'assemblage peut accroître la maintenabilité de la génération d'assemblage et faciliter le développement de nouveaux assemblages. C'est l'une des intentions de HLCM. Malheureusement, l'implémentation actuelle de HLCM ne passe pas à l'échelle et est trop lente pour les tailles d'architectures ciblées. Une nouvelle implémentation plus performante est en cours.

## 7.3 Discussion

En ce qui concerne les performances, les expérimentations montrent que :
- L$^2$C et les assemblages de FFT 3D passent à l'échelle jusqu'à 8192 cœurs ;
- Les assemblages L$^2$C bénéficient d'un équilibrage de charge sur des architectures hétérogènes là ou 2DECOMP est limité par la grappe de machine la plus lente ;
- Les assemblages utilisant la décomposition 1D sont compétitifs avec la FFTW et 2DECOMP
- Les assemblages utilisant la décomposition 2D sont un peu plus lents que 2DECOMP sur de grandes matrices (pas entièrement optimisé), mais compétitifs sur de petites matrices ; ils passent à l'échelle et bénéficient de l'équilibrage de charge.
- Les performances des assemblages dépendent de paramètres tel que le découpage des données qui peuvent être choisis automatiquement.

Les résultats sont encourageants, mais le nombre de cœurs utilisés reste encore petit face aux architectures ciblées dans les articles de recherche (*e.g.* 65536 cœurs pour P3DFFT [20]). Pour remédier à cela, nous travaillons actuellement sur l'amélioration de la phase de déploiement de $L^2C$ pour pouvoir mettre en place des expériences utilisant plus de cœurs sur les supercalculateurs Curie et Jade.

En ce qui concerne l'adaptation, l'approche par composants permet de produire simplement des assemblages spécialisés. De nombreuses optimisations d'articles de recherche ont été mis en œuvres, profitant ainsi de la réutilisation du code, du remplacement de composants et de l'adaptation d'attributs de composants. D'autre optimisations requièrent l'implémentation de nouveaux composants. Le processus de spécialisation permet de réutiliser la plupart des composants de base (de 69% à 100%) sans aucune modification.

# 8 Conclusion et perspectives

Le travail effectué durant ce stage consistait à sélectionner des algorithmes de FFTs en trois dimensions existants dans la littérature, à les implémenter dans le modèle de composants $L^2C$ et à évaluer leur performances, leur passage à l'échelle, leur réutilisation ainsi qu'à évaluer les performances des variantes les assemblages de composants résultants.

Les expériences sur Grid'5000 et Curie montrent que les assemblages $L^2C$ passent à l'échelle jusqu'à 8192 cœurs et sont compétitifs avec les bibliothèques existantes dans le cas homogène avec une décomposition 1D et dans le cas hétérogène avec des décompositions 1D ou 2D. Les mesures de performances montrent aussi qui est possible d'ajuster automatiquement certains paramètres de l'assemblage. Cependant, les assemblages basés sur une décomposition 2D nécessitent d'être encore un peu optimisés (notamment au niveau de la transposition). Les résultats de la réutilisation montrent que l'utilisation des composants permet d'écrire des applications optimisées en réutilisant des parties d'autres versions.

Des travaux sont encore nécessaires sur $L^2C$ et HLCM afin de pouvoir mettre en place des applications basées sur des composants passant à l'échelle sur des architectures possédant un grand nombre de cœurs. Malgré cela, à partir des nombreuses variantes d'assemblages $L^2C$ qui ont été implémentées, une description d'assemblage avec plus haut niveau d'abstraction pourrait être conçue en utilisant un modèle de composants de haut niveau tel que HLCM et profiterait ainsi des avantages liés à la hiérarchie, aux connecteurs et à la généricité des composants. Des algorithmes de choix pourraient ensuite être implémentés afin d'automatiser la spécialisation d'assemblage de FFT 3D selon l'architecture matérielle sous-jacente et les paramètres applicatifs. Un tel processus permettrait d'obtenir des performances portables à moindre coût sur un large ensemble d'architectures matérielles.

# Annexes

| Site | Grappe | Nœuds | CPU/Nœud | Cœurs/CPU | CPU | Freq. | Réseau |
|---|---|---|---|---|---|---|---|
| Grenoble | Adonis | 10 | 2 | 4 | Intel Xeon E5520 | 2.27 GHz | InfiniBand 40G |
| | Edel | 72 | 2 | 4 | Intel Xeon E5520 | 2.27 GHz | InfiniBand 40G |
| | Genepi | 34 | 2 | 4 | Intel Xeon E5420 QC | 2.5 GHz | InfiniBand 20G |
| Luxembourg | Granduc | 22 | 2 | 4 | Intel Xeon L5335 | 2.0 GHz | Ethernet 1G/10G |
| | Petitprince | 16 | 2 | 6 | Intel Xeon E5-2630L | 2.0 GHz | Ethernet 10G |
| Lyon | Hercule | 4 | 2 | 6 | Intel Xeon E5-2620 | 2.0 GHz | Ethernet 10G |
| | Orion | 4 | 2 | 6 | Intel Xeon E5-2630 | 2.3 GHz | Ethernet 10G |
| | Sagittaire | 79 | 2 | 1 | AMD Opteron 250 | 2.4 GHz | Ethernet 1G |
| | Taurus | 16 | 2 | 6 | Intel Xeon E5-2630 | 2.3 GHz | Ethernet 10G |
| Nancy | Graphene | 144 | 1 | 4 | Intel Xeon X3440 | 2.53 GHz | Infiniband 20G |
| | Graphite | 4 | 2 | 8 | Intel Xeon E5-2650 | 2.0 GHz | Ethernet 10G |
| | Griffon | 92 | 2 | 4 | Intel Xeon L5420 | 2.5 Ghz | Infiniband 20G |
| Nantes | Econome | 18 | 2 | 8 | Intel Xeon E5-2660 | 2.2 GHz | Ethernet 10G |
| Reims | StRemi | 44 | 2 | 12 | AMD Opteron 6164 HE | 1.7 GHz | Ethernet 1G |
| Rennes | Paradent | 64 | 2 | 4 | Intel Xeon L5420 | 2.5 GHz | Ethernet 1G |
| | Paranoia | 8 | 2 | 10 | Intel Xeon E5-2660v2 | 2.2 GHz | Ethernet 10G |
| | Parapide | 25 | 2 | 4 | Intel Xeon X5570 | 2.93 GHz | Infiniband |
| | Parapluie | 40 | 2 | 12 | AMD Opteron 6164 HE | 1.7 GHz | Infiniband |
| Sophia | Sol | 50 | 2 | 2 | AMD Opteron 2218 | 2.6 GHz | Ethernet |
| | Suno | 45 | 2 | 4 | Intel Xeon E5520 | 2.26 GHz | Ethernet |
| Toulouse | Pastel | 140 | 2 | 2 | AMD Opteron 248 | 2.2 GHz | Ethernet 1G |

FIGURE 8.1 – Liste des grappe de serveurs utilisés dans la section 7.1 afin dévaluer les performances de quelques assemblages décrits au chapitre 6.

# Bibliographie


[1] Frédéric Desprez, Geoffrey Fox, Emmanuel Jeannot, Kate Keahey, Michael Kozuch, David Margery, Pierre Neyron, Lucas Nussbaum, Christian Pérez, Olivier Richard, Warren Smith, Gregor Von Laszewski, and Jens Vöckler. Supporting Experimental Computer Science. Rapport de recherche RR-8035, INRIA, Jul 2012.

[2] M. Frigo and S.G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2) :216–231, February 2005.

[3] Jeffrey M Squyres and Andrew Lumsdaine. The Component Architecture of Open MPI : Enabling Third-Party Collective Algorithms. In *Component Models and Systems for Grid Applications*, pages 167–185. Springer, 2005.

[4] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition edition, 2002.

[5] Julien Bigot and Christian Pérez. High Performance Composition Operators in Component Models. In *High Performance Computing : From Grids and Clouds to Exascale*, volume 20 of *Advances in Parallel Computing*, pages 182 – 201. IOS Press, 2011.

[6] M. Bozga, M. Jaber, and J. Sifakis. Source-to-Source Architecture Transformation for Performance Optimization in BIP. *Industrial Informatics, IEEE Transactions on*, 6(4) :708–718, Nov 2010.

[7] Julien Bigot, Zhengxiong Hou, Christian Pérez, and Vincent Pichon. A low level component model easing performance portability of HPC applications. *Computing*, November 2013.

[8] Message Passing Interface Forum. MPI : A Message-Passing Interface Standard Version 3.0. 09 2012.

[9] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, 21(9) :948–960, sep 1972.

[10] Ralph Duncan. A Survey of Parallel Computer Architectures. *Computer*, 23(2) :5–16, February 1990.

[11] Top500. Top 500 Supercomputer. http ://www.top500.org/.

[12] IEEE. *IEEE 1003.1c-1995 : Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2 : Threads Extension (C Language)*. 1995.

[13] Leonardo Dagum and Ramesh Menon. OpenMP : An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1) :46–55, January 1998.

[14] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[15] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++ : A Portable Concurrent Object Oriented System Based on C++. *SIGPLAN Not.*, 28(10) :91–108, October 1993.

[16] National Institute of Standards and Technology (NIST). SPMD. http://xlinux.nist.gov/dads//HTML/singleprogrm.html, 1999.

[17] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[18] J.E. Stone, D. Gohara, and Guochun Shi. OpenCL : A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3) :66–73, May 2010.

[19] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC — First Experiences with Real-World Applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.

[20] Dmitry Pekurovsky. P3DFFT : A Framework for Parallel Computations of Fourier Transforms in Three Dimensions. *SIAM J. Scientific Computing*, 34(4), 2012.

[21] Roland Schulz. 3D FFT with 2D decomposition. CS project report http://cmb.ornl.gov/Members/z8g/csproject-report.pdf, April 2008.

[22] R. C. Le Bail. Use of Fast Fourier Transforms for Solving Partial Differential Equations in Physics. *J. Comput. Phys.*, 9(3) :440–65, 1972.

[23] Daniel Guinier. The Multiplication of Very Large Integers Using the Discrete Fast Fourier Transform. *SIGSAC Rev.*, 9(3) :26–27, June 1991.

[24] James Cooley and John Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90) :297–301, 1965.

[25] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, and Derrick Weathersby. Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems. In *SPAA*, pages 298–309, 1994.

[26] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler. Bandwidth-optimal All-to-all Exchanges in Fat Tree Networks. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, pages 139–148. ACM, Jun. 2013.

[27] Jeffrey M. Squyres and Andrew Lumsdaine. The Component Architecture of Open MPI : Enabling Third-Party Collective Algorithms. In Vladimir Getov and Thilo Kielmann, editors, *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185, St. Malo, France, July 2004. Springer.

[28] Rajeev Thakur and Rolf Rabenseifner. Optimization of Collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19 :49–66, 2005.

[29] N. Li and S. Laizet. 2DECOMP&FFT – A highly scalable 2D decomposition library and FFT interface. In *Cray User Group 2010 conference*, Edinburgh, 2010.

[30] Krishna Chaitanya Kandalla, Hari Subramoni, Karen A. Tomko, Dmitry Pekurovsky, Sayantan Sur, and Dhabaleswar K. Panda. High-performance and scalable non-blocking all-to-all with collective offload on InfiniBand clusters : a study with parallel 3D FFT. *Computer Science - R&D*, 26(3-4) :237–246, 2011.

[31] R. Agarwal and J. Cooley. New Algorithms for Digital Convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 25(5) :392–410, 1977.

[32] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[33] Brian Gough. *GNU Scientific Library Reference Manual - Third Edition*. Network Theory Ltd., 3rd edition, 2009.

[34] Jianxin Xiong, Jeremy Johnson, Robert W. Johnson, and David Padua. SPL : A Language and Compiler for DSP Algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001.

[35] Andreas Bonelli, Franz Franchetti, Juergen Lorenz, Markus Püschel, and Christoph W. Ueberhuber. Automatic Performance Optimization of the Discrete Fourier Transform on Distributed Memory Computers. In *International Symposium on Parallel and Distributed Processing and Application (ISPA)*, volume 4330 of *Lecture Notes In Computer Science*, pages 818–832. Springer, 2006.

[36] M. D. McIlroy. Mass-produced Software Components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[37] Don Batory. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1 :355–398, 1992.

[38] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL Component Model and Its Support in Java : Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12) :1257–1284, September 2006.

[39] Julien Bigot. *Du support générique d'opérateurs de composition dans les modèles de composants logiciels, application au calcul scientifique.* PhD thesis, INSA de Rennes, December 2010.

[40] Julien Bigot and Christian Pérez. Increasing Reuse in Component Models through Genericity. In *Proceedings of the 11th International Conference on Software Reuse, ICSR '09*, pages 21–30, Falls Church, VA, United States, September 2009. Springer-Verlag.

[41] Juergen Boldt. The Common Object Request Broker : Architecture and Specification. July 1995.

[42] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM : a grid extension to Fractal for autonomous distributed components. *Annales des Télécommunications*, 64(1-2) :5–24, 2009.

[43] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, HPDC '99, pages 13–, Washington, DC, USA, 1999. IEEE Computer Society.

[44] Bernholdt D.E., Allan B.A., Armstrong R., Bertrand F., Chiu K., Dahlgren T.L., Damevski K., Ewasif W.R., Epperly T.G.W, Govindaraju M., Katz D.S., Kohl J.A., Krishnan M., Kumfert G., Larson J.W., Lefantzi S., Lewis M.J., Malony A.D., McInnes L.C., Nieplocha J., Norris B., Parker S.G., J. Shende Ray, T.L. S. Windus, and S Zhou. A Component Architecture for High Performance Scientific Computing. *International Journal of High Performance Computing Applications*, May 2006.

[45] Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lèbre, and Takahiro Hirofuchi. Using the EXECO toolbox to perform automatic and reproducible cloud experiments. In *1st International Workshop on UsiNg and building ClOud Testbeds (UNICO, collocated with IEEE CloudCom 2013*, Bristol, Royaume-Uni, Sep 2013.